Model-Driven Software Engineering for Virtual Machine Images Provisioning in Cloud Computing

Tam Le Nhan

ANNÉE 2013

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Nhan Tam LE

préparée à l'INRIA
Institut National de Recherche en Informatique et Automatique
Centre Rennes Bretagne Atlantique

Model-Driven Software Engineering for Virtual Machine Images Provisioning in Cloud Computing

Thèse soutenue à Rennes le 10 Décembre 2013

devant le jury composé de :

Jean-Louis PAZAT
Professeur, INSA de Rennes
Président

Jean-Marc MENAUD
Maître de conférence, Ecole des Mines de Nantes
Rapporteur

Philippe COLLET
Professeur, Université de Nice
Rapporteur

Jean-Marc JÉZÉQUEL
Professeur, Université de Rennes 1
Directeur de thèse

Gerson SUNYÉ
Maître de conférence, Université de Nantes
Encadrant

Acknowledgments

First of all, I would like to sincerely thank to my advisors, Prof. Jean-Marc Jézéquel and Dr. Gerson Sunyé for guiding and supporting me at every step throughout my Ph.D, for everything I learned about research from them. I specially thank Prof. Jean-Marc Jézéquel for giving me the opportunity to come to France for doing my Ph.D in INRIA, the excellent research institute in computer science that I have known.
I would like to special thank to reviewers, Dr. Jean-Marc Menaud and Prof. Philippe Collet for spending time to review my thesis and give me the valuable comments on it. I also would like to thanks Prof. Jean-Louis Pazat for examining my thesis.
Many thanks to every members of Triskell team for the discussions, collaborations and for all the good moments after work. Thanks to all my Vietnamese friends for sharing, and supporting me during three years of my Ph.D in Rennes.
I also want to thank the French National Institute for Research in Computer Science and Control (INRIA) and Vietnamese Government for the funding of my Ph.D work.
Finally, my deepest gratitude goes towards my parents, my wife, my sons and everyone in my big family. They are the ones who always follow and support me unconditionally and in every moment of my life.

Abstract

The Cloud Computing Infastructure-as-a-Service (IaaS) layer provides a service for on demand virtual machine images (VMIs) deployment. This service provides a flexible platform for cloud users to develop, deploy, and test their applications. The deployment of a VMI typically involves booting the image, installing and configuring the software packages. In the traditional approach, when a cloud user requests a new platform, the cloud provider selects an appropriate template image for cloning and deploying on the cloud nodes. The template image contains pre-installed software packages. If it does not fit the requirements, then it will be customized or the new one will be created from scratch to fit the request.
In the context of cloud service management, the traditional approach faces the difficult issues of handling the complexity of interdependency between software packages, scaling and maintaining the deployed image at runtime. The cloud providers would like to automate this process to improve the performance of the VMIs provisioning process, and to give the cloud users more flexibility for selecting or creating the appropriate images while maximizing the benefits for providers intern of time, resources and operational cost.
This thesis proposes an approach to manage the interdependency of the software packages, to model and automate the VMIs deployment process, to support the VMIs reconfiguration at runtime, called the *Model-Driven approach*. We particularly address the following challenges: (1) modeling the variability of virtual machine image configurations; (2) reducing the amount of data transfer through the network; (3) optimizing the power consumption of virtual machines; (4) easy-to-use for cloud users; (5) automating the deployment of VMIs; (6) supporting the scaling and reconfiguration of VMIs at runtime; (7) handling the complex deployment topology of VMIs.
In our approach, we use Model-Driven Engineering techniques to model the abstraction representations of the VMI configurations, the deployment and the reconfiguration processes of virtual machine image. We consider the VMIs as a product line and use the feature models to represent the VMIs configurations. We also define the deployment, re-configuration processes and their factors (e.g. virtual machine images, software packages, platform, deployment topology, etc.) as the models. On the other hand, the *model-driven approach* relies on the high-level abstractions of the VMIs configuration and the VMIs deployment to make the management of virtual images in the provisioning process to be more flexible and easier than traditional approaches.

Keywords:

Cloud Computing, Virtual Machine Images Provisioning, Model-Driven Engineering, Product Lines Engineering, Feature Model, Mode@Runtime.

Résumé en français

Le contexte et la problématique

De nos jours, le cloud computing est omniprésent dans la recherche et aussi dans l'industrie. Il est considéré comme une nouvelle génération de l'informatique où les ressources informatiques virtuelles à l'échelle dynamique sont fournies comme des services via l'internet. Les utilisateurs peuvent accéder aux systèmes de cloud utilisant différentes interfaces sur leurs différents dispositifs. Ils ont seulement besoin de payer ce qu'ils utilisent, respectant le l'accord de service (Service-Layer Agreement) établi entre eux et les fournisseurs de services de cloud.

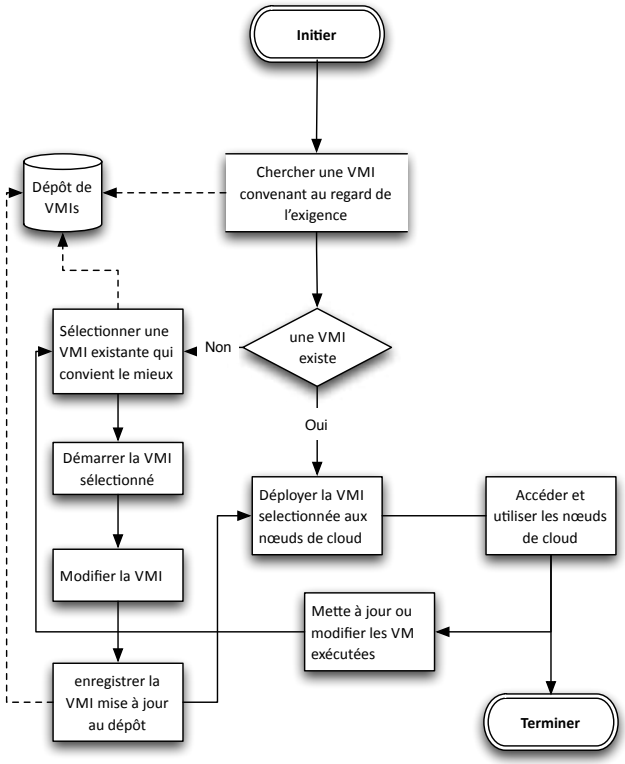


Figure 1: Le processus de provisionnement VMI traditionnelle

Une des caractéristiques principales du cloud computing est la virtualisation grâce à laquelle toutes les ressources deviennent transparentes aux utilisateurs. Les utilisateurs n'ont plus besoin de contrôler et de maintenir les infrastructures informatiques. La virtualisation dans le cloud computing combine des images de machines virtuelles (VMIs) et des machines physiques où ces images seront déployées. Typiquement, le déploiement d'une telle VMI comprend le démarrage de l'image, l'installation et la configuration des packages définis pas la VMI. Dans les approches

traditionnelles, les VMIs sont crées par les experts techniques des fournisseurs de services cloud. Il s'agit des VMIs pré-packagés qui viennent avec des composants pré-installés et pré-configurés. Pour répondre à une requête d'un client, le fournisseur sélectionne une VMI appropriée pour cloner et déployer sur un nœud de cloud. Si une telle VMI n'existe pas, une nouvelle VMI va être créée pour cette requête. Cette VMI pourrait être générée à partir de la VMI existante la plus proche ou être entièrement neuve. Le cycle de vie de l'approvisionnement d'une VMI dans l'approche traditionnelle est décrite dans la Figure 1.
Une VMI standard contient normalement plusieurs packages parmi lesquels certains qui ne seront jamais utilisés. Cela vient du fait que la VMI est créée au moment de conception pour le but d'être clonée plus tard. Cette approche a des inconvénients tels que la demande de ressources importantes pour stocker des VMIs ou pour les déployer. De plus, elle requiert le démarrage de plusieurs composants, y compris ceux non utilisés. Particulièrement, à partir du point de vue de gestion de services, il est difficile de gérer la complexité des interdépendances entre les différents composants afin de maintenir les VMIs déployées et de les faire évoluer.
Pour résoudre les problèmes énumérés ci-dessus, les fournisseurs de services de cloud pourraient automatiser le processus d'approvisionnement et permettre aux utilisateurs de choisir des VMIs d'une manière flexible en gardant les profites des fournisseur en terme de temps, de ressources, et de coût. Dans cette optique, les fournisseurs devraient considérer quelques préoccupations: (1) Quels packages et dépendances seront déployés? (2) Comment optimiser une configuration en terme de coût, de temps, et de consommation de ressources? (3) Comment trouver la VMI la plus ressemblante et comment l'adapter pour obtenir une nouvelle VMI? (4) Comment éviter les erreurs qui viennent souvent des opérations manuelles? (5) Comment gérer l'évolution de la VMI déployée et l'adapter aux besoins de reconfigurer et de passer automatiquement à l'échelle? A cause de ces exigences, la construction d'un systèmes de gestion de plateformes cloud (PaaS – Platform as a Sevice) est difficile, particulièrement dans le processus d'approvisionnement de VMIs. Cette difficulté requiert donc une approche appropriée pour gérer les VMIs dans les systèmes de cloud computing. Cette méthode fournirait des solutions pour la reconfiguration et le passage automatique à l'échelle.

Les défis et les problèmes clés

A partir de la problématique, nous avons identifié sept défis pour le développement d'un processus d'approvisionnements dans cloud computing.

- **C1**: *Modélisation de la variabilité des options de configuration des VMIs afin de gérer les interdépendances entre les packages logiciels*
  Les différents composants logiciels pourraient requérir des packages spécifiques ou des bibliothèques du système d'exploitation pour une configuration correcte. Ces dépendances doivent être arrangées, sélectionnées, et résolues manuellement pour chaque copie de la VMI standard. D'autre part, les VMIs sont créées pour répondre aux exigences d'utilisateurs qui pourraient partager des sous-besoins en commun. La modélisation de la similitude et de la variabilité des VMIs au regard de ces exigences est donc nécessaire.

- **C2**: *Réduction des données transférées via les réseaux pendant le processus d'approvisionnement*
  Afin d'être prêt pour répondre aux requêtes de clients, plusieurs packages sont installés sur la machine virtuelle standard , y compris les packages qui ne seront pas utilisé. Ces

packages devront être limités afin de minimaliser la taille des VMIs.

- **C3**: *Optimisation de la consommation de ressources pendant l'exécution*
  Dans l'approche traditionnelle, les activités de création et de mise à jour des VMIs requièrent des opérations manuelles qui prennent du temps. D'autre part, tous les packages dans les VMIs, y compris ceux qui ne sont pas utilisés, sont démarrés et occupent donc des ressources. Ces consommations de ressources devraient être optimisées.

- **C4**: *Mise à disposition d'un outil interactif facilitant les choix de VMIs des utilisateurs*
  Les fournisseurs de services cloud voudraient normalement donner la flexibilité aux utilisateurs clients dans leurs choix de VMIs. Cependant, les utilisateurs n'ont pas de connaissances techniques approfondies. Pour cette raison, des outils facilitant les choix sont nécessaires.

- **C5**: *Automatisation du déploiement des VMIs*
  Plusieurs opérations du processus d'approvisionnement sont très complexes. L'automatisation de ces opérations peut réduire le temps de déploiement et les erreurs.

- **C6**: *Support de la reconfiguration de VMIs pendant leurs exécutions*
  Un des caractéristiques importantes de cloud computing est de fournir des services à la demande. Puisque les demandes évoluent pendant l'exécution des VMIs, les systèmes de cloud devraient aussi s'adapter à ces évolutions des demandes.

- **C7**: *Gestion de la topologie de déploiement de VMIs*
  Le déploiement de VMIs ne doit pas seulement tenir en compte multiple VMIs avec la même configuration, mais aussi le cas de multiple VMIs ayant différentes configurations. De plus, le déploiement de VMIs pourrait être réalisé sur différentes plateformes de cloud quand le fournisseur de service accepte une infrastructure d'un autre fournisseur

Afin d'adresser ces défis, nous considérons trois problèmes clés pour le déploiement du processus d'approvisionnement de VMIs:

1. *Besoin d'un niveau d'abstraction pour la gestion de configurations de VMIs*: Une approche appropriée devrait fournir un haut niveau d'abstraction pour la modélisation et la gestion des configurations des VMIs avec leurs packages et les dépendances entre ces packages. Cette abstraction permet aux ingénieurs experts des fournisseurs de services de cloud à spécifier la famille de produits de configurations de VMIs. Elle facilite aussi l'analyse et la modélisation de la similitude et de la variabilité des configurations de VMIs, ainsi que la création des VMIs valides et cohérentes.

2. *Besoin d'un niveau d'abstraction pour le processus de déploiement de VMIs*: Une approche appropriée pour l'approvisionnement de VMIs devrait fournir une abstraction du processus de déploiement.

3. *Besoin d'un processus de déploiement et de reconfiguration automatique*: Une approche appropriée devrait fournir une abstraction du processus de déploiement et de reconfiguration automatique. Cette abstraction facilite la spécification, l'analyse, et la modélisation la modularité du processus. De plus, l'approche devrait supporter l'automatisation afin de réduire les tâches manuelles qui sont couteuses en terme de performance et contiennent potentiellement des erreurs.

Présentation de la solution

Cette thèse propose une approche de la gestion de VMI pour les environnements de Cloud Computing, fournissant une façon de s'adapter aux besoins des images de machine virtuelle de l'auto-mise à l'échelle et de l'auto-configuration, appelée approche pilotée par les modèles. Dans cette approche, nous utilisons l'approche de l'Ingénierie Dirigée par les Modèles (IDM) pour manipuler les configurations VMI et le processus de déploiement automatisé de VMIs dans l'environnement de cloud computing. Nous considérons les VMIs comme une ligne de produit et utilisons des modèles de caractéristiques à représenter les configurations de VMI et des techniques basées sur des modèles pour manipuler le déploiement automatique VMI et reconfiguration. Le modèle de caractéristiques est utilisé pour traiter les caractéristiques avec des attributs (par exemple, le temps d'installation, de la taille du logiciel, etc.) et de renforcer le processus de raisonnement pour trouver les configurations optimales de VMI. En ajoutant les attributs des caractéristiques, nous pouvons évaluer et trouver la configuration optimale en fonction de chaque critère, tels que le temps d'installation minimale ou la taille de VMI. En outre, l'utilisation de l'approche pilotée par les modèles permet de réduire la consommation d'énergie et de s'adapter aux besoins des images de machine virtuelle de l'auto-mise à l'échelle et de la reconfiguration.

Modélisation de caractéristiques pour la gestion des configurations VMI

Dans notre approche, les lignes de produits de configuration VMI sont décrites en utilisant des modèles de caractéristiques (feature models). En termes de dérivation de configuration VMI, un modèle de caractéristiques décrit:

- Les progiciels qui sont nécessaires pour composer une image de machine virtuelle, représentée comme des options de configuration.
- Les règles dictant les exigences, telles que les paquets dépendants et les bibliothèques requises par chaque composant de logiciel.
- Les contraintes, qui spécifient comment le choix d'un composant donné restreint le choix des autres composants, dans la même image de machine virtuelle.

L'approche de modélisation de caractéristiques porte sur deux modèles: le modèle de caractéristiques VMI, le modèle résolu VMI et un processus de dérivation du produit (Figure 2)

- *Le modèle de caractéristiques VMI*: représente la ligne de produits entière avec toutes ses caractéristiques comme les options de configuration, de leurs relations et des contraintes qui pourraient être utilisées pour composer une VMI.
- *Le modèle résolu VMI*: est dérivé du processus de dérivation du produit en fonction des sélections de l'utilisateur sur le modèle de la fonction VMI. Il inclut les caractéristiques sélectionnées et leurs dépendances.
- *Le processus de dérivation de produit*: génère les configurations VMI de la combinaison du modèle de caractéristiques VMI et les sélections de l 'utilisateur.

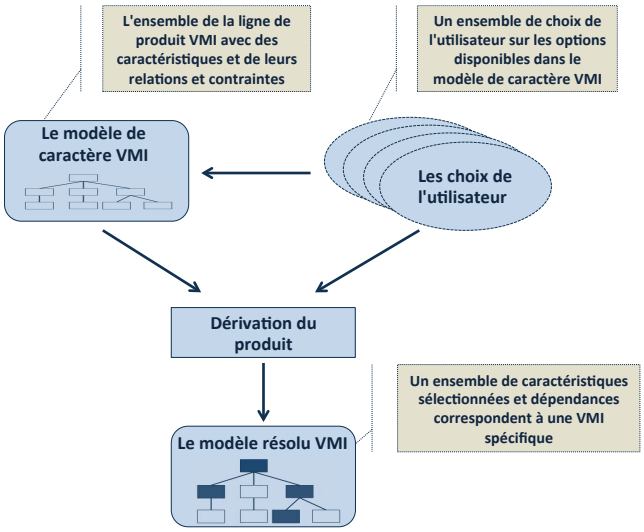


Figure 2: Une architecture globale de l'approche de modélisation de caractéristiques

Nous utilisons une bibliothèque de source libre pour le raisonnement automatisé sur les modèles de caractéristiques (SPLAR). Il a été développé par Marcilio Mendoca dans son travail de thèse. Cette bibliothèque propose des composants SAT et est basée sur BDD pour raisonner et pour configurer des modèles de caractéristiques. Elle fournit un support de validation des les sélections de caractéristiques, et à générer les configurations valides à partir des caractéristiques et des dépendances sélectionnés. Il propose une contrainte spécifique au domaine solveur appelé le système de contraintes d' arbre de caractéristiques qui adapte les algorithmes de raisonnement pour les arbres de caractéristiques.
Cependant, le moteur SPLAR fournit seulement un soutien pour le raisonnement sur les arbres de caractéristiques; avec des caractéristiques contenant deux attributs de base: ID et nom tandis que dans notre approche, un caractéristique sur l'arbre de caractéristiques représente une option de configuration (un progiciel). Il contient des informations utilisées pour l'optimisation de la configuration d'une image de machine virtuelle, tels que: le temps d'installation, le temps de désinstallation, la taille du paquet, coût, etc. Par conséquence, afin d'utiliser SPLAR comme un moteur de raisonnement pour les modèles de caractéristiques VMI, nous étendons le méta-modèle original des modèles de caractéristiques prises en charge par le moteur SPLAR pour représenter les modèles de caractéristiques VMI, et nous améliorons SPLAR pour permettre la recherche de la configuration optimale des informations supplémentaires basées VMI sur les caractéristiques.

L'ingénierie dirigée par les modèles pour le déploiement et la reconfiguration dynamique des VMIs

Les approches dirigées par les modèles favorisent la création et l'utilisation des modèles de domaines. Un tel modèle fournit une abstraction représentant les connaissances et les activités qui gèrent un domaine d'application particulière. Dans le contexte du processus d'approvisionnement

des VMIs dans cloud computing, l'approche dirigée par les modèles fournit une utilisation systématique des modèles en tant que des artefacts élémentaires au long du déploiement, l'installation, et la reconfiguration des images et des packages.

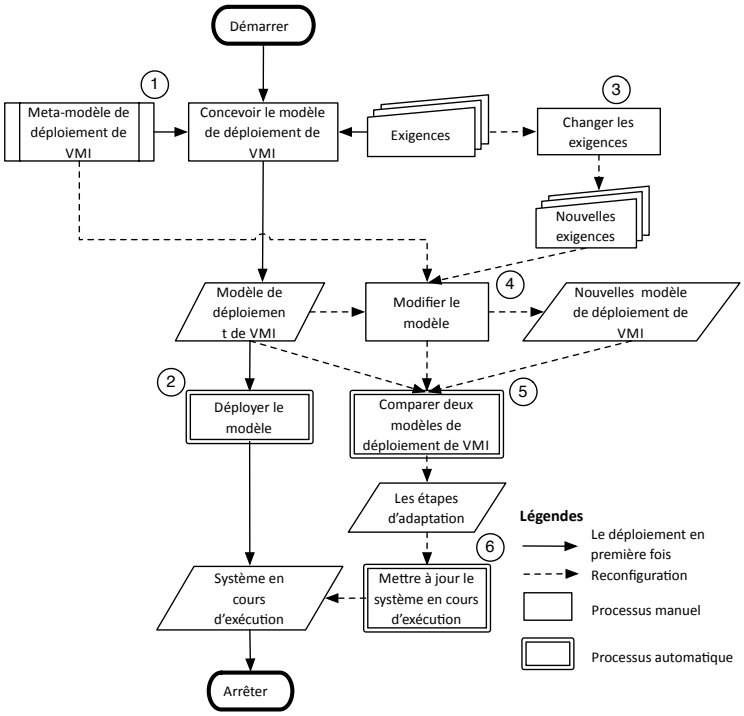


Figure 3: Le déploiement et la reconfiguration pendant l'exécution des VMIs basés sur les modèles

Dans l'approche traditionnelle, les composants logiciels sont assemblés en même temps que la VMI est créée. En revanche, dans l'approche dirigée par les modèles, les composants sont assemblés au démarrage. La Figure 3 montre le processus de déploiement et de reconfiguration. Le processus contient deux types de sous-processus : manuel ((1), (3), et (4)) et automatique ((2), (5), (6)). Dans le processus (1), après avoir analysé les exigences d'utilisateurs, le fournisseur de services cloud crée une configuration appropriée et un modèle de déploiement conforme au méta-modèle de déploiement de VMIs. La sortie de ce processus est un modèle de déploiement de VMIs. Il contient une présentation du logiciel à installer sur une VMI et aussi les connexions du logiciel installé sur différentes VMIs. Dans le processus (2), le modèle de déploiement de VMIs créé est déployé pour créer un système avec les machines virtuelles espérées. Dans ce processus, les images de template initiales avec les systèmes opérations appropriées (en fonction des configurations définies dans l'étape précédente) are démarrées aux nœuds de cloud. Quand le démarrage est accompli, le modèle de déploiement de VMIs est déployé et exécuté directement sur les VMIs en cours d'exécution. L'installation et la configuration des composants logiciels ont aussi lieu dans ces VMIs en cours d'exécution.
Pendant l'exécution, la modification de ces VMIs est implémentée par le changement de la con-

figuration de VMIs dans le modèle de déploiement de VMIs et le redéploiement du nouveau modèle de déploiement aux VMIs en cours d'exécution. Le changement d'exigences d'utilisateurs (le processus (3)) ramène à la création d'un nouveau modèle de déploiement (le processus (4)). Le nouveau modèle est dérivé à partir de celui existant en fonction du changement d'exigences et conforme au méta-modèle de déploiement. Le nouveau modèle est comparé à celui existant pour déterminer les différences et proposer les étapes d'adaptation (le processus (5)). Finalement, ces étapes sont appliquées au systèmes en cours d'exécution (le processus (6)) pour le changer vers le nouveau système avec les machines virtuelles espérées.

Conclusion

Dans ce travail, nous avons proposé une approche dirigée par les modèles pour l'approvisionnement d'images de machines virtuelles dans cloud computing. Nous avons utilisé la technique de modélisation des caractéristiquess pour gérer les configurations des machines virtuelles, et puis, nous avons utilisé les techniques basées sur les modèles pour modéliser et définir les processus de déploiement et de reconfiguration des VMIs. Nous avons montré que notre approche améliore la performance du processus de provisionnement et rend la gestion des VMIs plus flexible et plus facile par rapport à l'approche traditionnelle. Nous avons aussi présenté l'originalité de notre approche. Particulièrement, les différences entre notre approches et celle traditionnelle sont est la suivante:

- **L'approche traditionnelle**: Une VMI standard contenant tous les packages possibles peut être clonée et démarrée plusieurs fois.
- **L'approche dirigée par les modèles**: Les VMIs, les packages, et les topologies de déploiement are considérés comme des modèles ; la configuration des VMIs sont créées au moment de conception, tant dis que les VMIs concrètes sont créées dynamiquement pendant l'exécution.

Nous avons aussi réalisé quelques travaux empiriques pour évaluer notre approche. Nous avons expérimenté en utilisant deux environnement de virtualisation, y compris Amazon Elastic Compute Cloud et Grid5000. L'objectif de ces expérimentations est d'évaluer l'avantage de notre approche dirigée par les modèles et de montrer comment l'approche répond aux défis adressés.

Contents

List of Figures

List of Tables

Part I

Introduction & State of The Art

Chapter 1

Introduction

Contents

1.1 Problem statement

Cloud Computing [7, 32] has been recently a hot topic in both research and industry. It can be described as a new kind of computing in which dynamically scalable and virtualized resources are provided as services over the Internet. Cloud users can access cloud system and use the service through different devices and interfaces. The users only have to pay what they use according to Service Level Agreement contracts established between Cloud providers and Cloud users [14]. One of the main features of Cloud computing is the virtualization in which all cloud resources become transparent to the user. Users do not need any longer to control and maintain the underlying cloud infrastructure. The virtualization in Cloud Computing combines a number of virtual machine images (VMIs) on the top of physical machines. Each virtual image hosts a complete software stack: it includes operating system, middleware, database, and development applications. The deployment of a VMI typically involves booting the image, as well as the installation and the configuration of software packages. In the traditional approach, the creation of a VMI to fit user 's requirements and its deployment in the Cloud environment are typically carried out by the technical division of the Cloud service providers who provide a platform as a service to the user according to SLA contracts signed between the service provider and the user. Usually, it is a pre-packaged VMI with installed and configured software components. When a cloud user requests a new platform, the service provider administrators select an appropriate VMI for cloning and deploying on cloud nodes. If there is no match found, then a new one

is created and configured to match the request. It can be generated by modifying from the closest-fit existing VMI or from scratch. The life-cycle of the traditional VMI provisioning process is described the detail in figure 1.1.

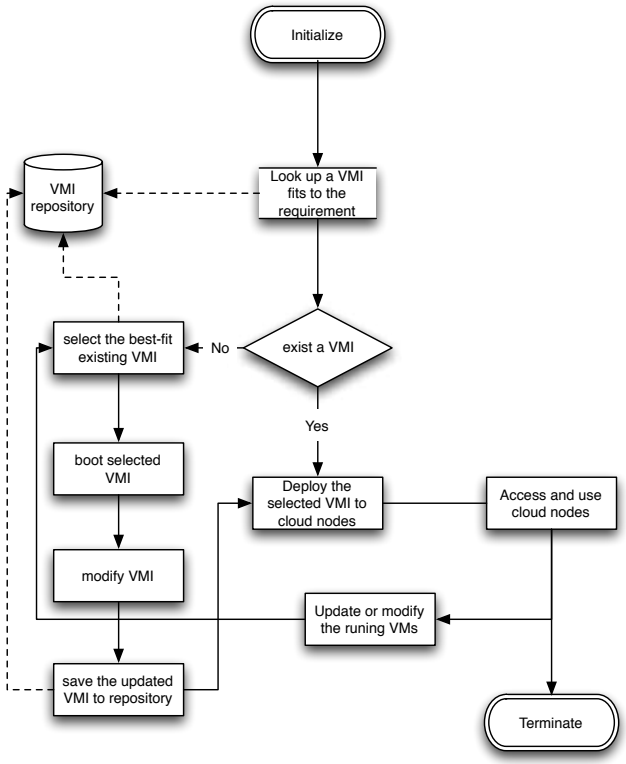


Figure 1.1: Life cycle of the Traditional VMI provisioning process

The standard VMI contains many software packages, which rarely get used and thus the image is typically larger than the necessary. Actually, It is created at design time and not used for execution, but it is considered as an image source for the cloning. The standard VMI will be replicated into new virtual images and deployed on the cloud nodes as one or more instances of standard virtual image. This can lead to several disadvantages, such as waste of storage space, memory, operating costs, CPU usage, and network bandwidth when cloning an image and deploying it on the cloud nodes [3]. It also requires more power consumption at runtime because the unneeded software start and run when the VMI is booted. Especially, from the point of view of service management, It is difficult to handle the complexity of interdependency between software components, to maintain the deployed VMIs at runtime, and scale service manually.

Cloud service providers would like to automate this process and give users more flexibility when choosing the appropriate VMI to satisfy their requirements [16], while ensuring benefits for providers in terms of time, operating costs, and resources. For the above reasons, there are several concerns would need to be addressed by the cloud providers in the building an automatic VMI provisioning process, such as: (i) Which software packages and their dependencies should

be installed? (ii) How to create an optimal configuration, in terms of saving operation cost, time or power consumption? (iii) How to find the best-fit existing VMI and how to obtain a new VMI by modifying this one? (iv) How to reduce the error-proneness from the manual operations. (v) How to handle the change of the deployed VMIs and adapt it to the needs of auto-scaling and re-configuring VMIs at run-time?

Because of these critical requirements, building an efficient PaaS cloud manager is challenging, particularly in the context of virtual machine image provisioning process. Therefore, it needs an appropriate approach for managing VMI for Cloud Computing environments, providing a way to adapt to the needs of auto-scaling and self-configuring virtual machine images.

1.2 Challenges and Key Issues

1.2.1 Challenges

From the problem statement, we determined eight challenges for the development of an efficient VMI provisioning process in cloud computing. The first four challenges are interrelated, they concern the VMI configurations management process. The fifth one is related to the easiness of the system. The last four challenges highlight different aspects needed to be considered for building automatic deployment and reconfiguration of VMIs at runtime.

- *C1 - Modeling the variability of VMI 's configuration options to handle the interdependencies of software packages:* Different software components may require specific packages or libraries in Operating System or other components in order to configure correctly. These interdependencies must be arranged, selected and resolved manually for every copy of the standard VMI. The VMIs are created to satisfy the requirement of cloud users. These requirements can have some common parts, and sometimes the selecting the software components can have more than one option. Therefore, it needs the modeling of commonality and variability of VMI 's configuration options.

- *C2 - Reducing the amount of data transferred through the networking in the provisioning processes:* To be ready to fulfill different requirements from users, the standard virtual machine is often installed many types of software depending on the purpose of using of the virtual machine. This means that many software packages are unneeded to the users will also have installed in the virtual machine, it leads to the size of the virtual machine images can be larger than necessary.

- *C3 - Optimizing the power consumption of VMIs at runtime:* In the traditional approach, every time a standard virtual image is created, or updated, it involves manual operations; it takes time for process of installing, customizing and configuring software packages. In addition, the standard virtual image could store too many unnecessary software packages. These packages are booted and run when the virtual machine runs, they also occupy memory, CPU and other resources. This leads the virtual machine to consume more power than necessary.

- *C4 - Providing the graphical interface and easy-to-use tools for user interactions:* Cloud providers want to give users more flexibility when choosing the appropriate VMI to satisfy their requirements. However, the cloud user does not have the deep technical knowledge

about software components, and underlying systems. Hence, the VMI provisioning process should be easy to use for cloud users.

- *C5 - Automating the deployment of VMIs:* Many operations of the VMI provisioning process are highly complex. Automating these operations could help to reduce the deployment time, cost and error-proneness.
- *C6 - Supporting the reconfiguration of VMIs at runtime:* One of the key features of cloud computing is the providing "Services on demand" while user 's requirements can be changed at runtime, so that the system needs to adapt to these changes and scale-up or scale-down and re-configure the running VMIs.
- *C7 - Handling the complex and flexible deployment topology of VMIs:* The deployment of virtual machine images should handle not only the multiple VMIs with the same configuration, but also handle the scenario that the virtual machine images have different configurations, for example the N-tiers web application scenario. Additionally, the VMIs deployment can be executed on the various cloud platforms when the service provider leases the infrastructure from other third-party service providers.

1.2.2 Key Issues

To address the challenges, we consider three key issues for the development of VMI provisioning process:

1. **Need of an abstraction level for VMI Configuration management:** The approach should provide a high level abstraction for modeling and managing the VMI 's configuration options (software packages and dependencies). This abstraction helps IT experts of cloud providers specify the product families of VMI configuration. It also helps analyzing, modeling the commonality and variability of VMI configurations, and creating the valid and consistent VMIs.
2. **Need of an abstraction level for VMI Deployment process:** An approach for VMI provisioning in cloud computing should provide an abstraction representation of the deployment process.
3. **Need of an automated deployment and re-configuration process:** The approach should provide an abstraction of the automated deployment and reconfiguration process. This abstraction helps for specifying, analyzing and modeling process modularity. Moreover, the approach should be highly-automated in order to reduce the manual tasks, error-proneness and improve the performance of VMI provisioning process.

These issues equate to limitations of the state-of-the-art addressed in this thesis. They will be used to analyze related work, describe the contribution and results.

1.3 Overview of The Solution

This thesis proposes an approach for managing VMI for Cloud Computing environments, providing a way to adapt to the needs of auto-scaling and self-configuring virtual machine images,

called Model-Driven approach. In this approach, we use Model-Driven Engineering (MDE) approach for managing VMI configurations and the automated deployment process of VMIs in cloud environment. We consider VMIs as a product line and use feature models to represent VMI configurations and model-based techniques to handle automatic VMI deployment and reconfiguration. Feature model is used to handle features with attributes (e.g., installation time, size of software, etc.) and enhance the reasoning process for finding the optimal configurations of VMI. By adding the attributes to the features, we can evaluate and find the optimal configuration according to each criterion, such as minimum installation time or VMI size. Moreover, applying the model-driven approach helps to reduce the power consumption and virtual machine image adaptation according to the needs of auto-scaling and self-configuration. The approach will be described in Chapter 3 and 4.

1.3.1 Feature Modeling for VMI Configuration Management

Our approach uses feature modeling to manage the configuration of virtual machine images. In terms of configuration derivation, a feature model describes:

- The software packages that are needed to compose a Virtual Machine Image, represented as configuration options.
- The rules dictating the requirements, such as dependent packages and the libraries required by each software component.
- The constraining rules, which specify how the choice of a given component restricts the choice of other components, in the same Virtual Machine Image.

Feature modeling approach deals with two models: Base Model and Resolved Model; and the Product Derivation Process.

- **VMI feature model:** represents configuration options and constraints which would be used for composing a VMI.
- **VMI resolved model:** is derived from product derivation process based on user 's selections on the VMI feature model. It includes the selected features and their dependencies.
- **Product derivation process:** generates the VMI configurations from the combination of VMI feature model and user 's selections.

1.3.2 Model-Based Deployment Process

Unlike the traditional approach, where software packages are installed and configured together when the VMI template is created, the model-driven deployment approach installs and configures software packages at deployment time, when a template VMI is booted. The approach also supports synchronization of maintenance of the deployed VMIs at runtime. In our approach, we create models that drive the creation of VMIs instances on demand. Every time a new virtual machine is created on a cloud node; the cloud provider selects features of VMI, generates configurations and applies the model to it. Actually, the model-based approach focuses on the modeling and systemizing the process of creating an virtual image. This approach abstracts the process from the virtual image, so creating multiple virtual images from applying a systemized

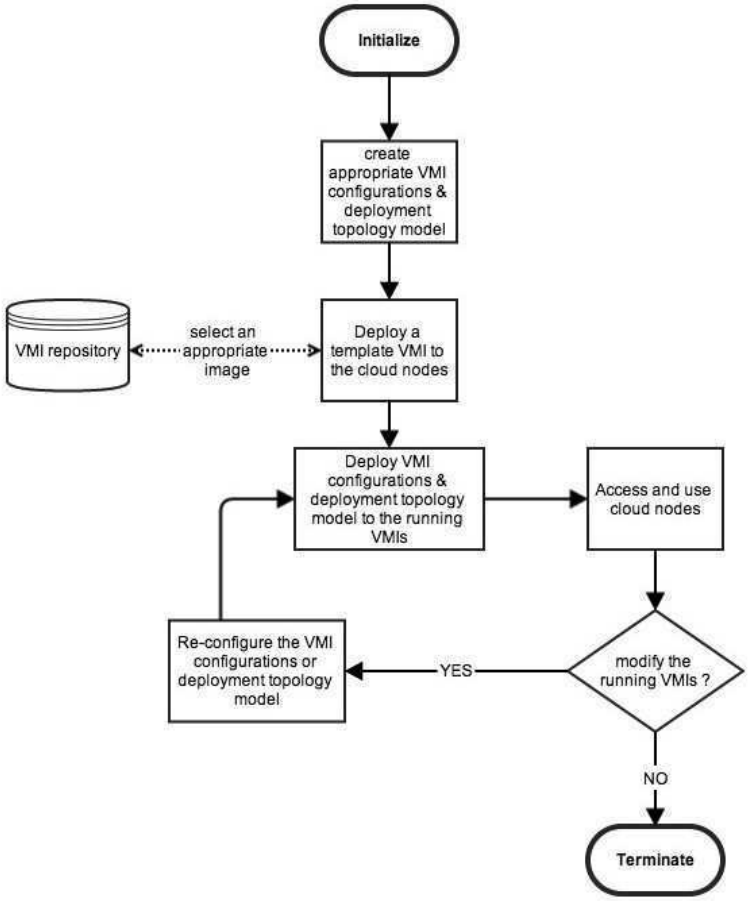


Figure 1.2: Life-cycle of the model-based VMI provisioning process

process to an initial virtual image rather than the time-consuming copying of a standard virtual image.

1.3.3 Claims

We claim that the MDE approach for developing a VMI provisioning in cloud computing enables to solve the seven challenges identified in Section 1.2.1. First, the use of the feature model for the representation of configuration options (software components) and their relationships support to handle the complexity of interdependencies of software packages. It also supports modeling the variability of VMI configurations (challenge C1). In our implementation, the VMI feature models are built by IT experts of cloud providers, who have knowledge about systems and software packages used to compose Virtual Machine Images; and many approaches and tools were proposed to automate analysis of the feature models [45, 10, 36]. They offer to validate, check satisfiability, detect the invalid features and analyze feature models. Second, the product derivation process helps to find the optimal configurations of VMIs according to the requirements. The created VMI configurations do not contain unneeded software. Therefore,

the virtual machines will consume less power consumption (challenge C3). It also reduces the size of the created VMIs, so the cloning the VMIs to the cloud node for deploying will consume less network bandwidth than in the traditional approach (challenge C2). Third, model-based deployment process supports to encapsulate of the deployment and management of the VMI provisioning process into a series of procedural operations. It helps to automate the VMI deployment process (challenge C5). Fourth, by dealing with the Resolved model (expected VMI configuration) which is independently of the standard VMIs, it makes easier to maintain and scaling deployed VMIs at the runtime (challenges C6 and C7). Finally, the VMI configuration manager supports the visual representation of software package and relationships, and the VMIs deployment manager which is developed based on Eclipse Modeling Framework provides graphical user interfaces. Thus, the users can create the VMI configurations and design the expected deployment scenario easily (challenge C4).

1.4 Contribution of The Thesis

The results of our work contributes to two key aspects of model-driven VMI provisioning process in cloud computing with the comparison to related work. By the survey of a number of related work in Chapter 2, we found that there are some different approaches that support some of the identified challenges. However, to the best of our knowledge, there is no existing approach that fulfills all the challenges which are identified in 1.2.1. We clarify our contribution of the thesis on two domains:

1.4.1 Contributions to the VMI configuration management

The contributions of Model-Driven approach for VMI provisioning process to the domain of VMI configuration management are:

- *An abstract representation* of software components and their relationships: This representation is called *feature modeling* for VMI configuration management. It helps to reduce the complexity of configuration management, reduce the error-proneness during the manipulation, and make cloud users easy to use.
- A prototype that provides the *feature model* to represent VMI 's configuration options and the product derivation process to generating appropriate VMI configurations.

1.4.2 Contributions to the VMI deployment and reconfiguration at runtime

The contributions of Model-Driven approach for VMI provisioning process to the domain of VMI deployment are:

- *An abstract modeling* of the VMI deployment process in cloud environments: This helps to encapsulate the deployment and management of the VMI provisioning process into a series of procedural operations.
- A prototype that supports the *model-based* deployment of VMIs in cloud environments: The prototype supports to automatic deploy, reconfigure and scaling VMIs at runtime.

1.5 Structure of The Thesis

The remainder of the thesis is organized as follows:

- **Chapter 2** presents the state of the art in virtual machine image provisioning in cloud computing.
- **Chapter 3** presents the feature modeling appproach for managing the VMI 's configuration options and deriving the appropriate VMI configurations.
- **Chapter 4** presents the model-based deployment and reconfiguration process of VMI provisioning in cloud computing.
- **Chapter 5** describes the experiments and evaluation of the approach.
- **Chapter 6** concludes the thesis and discusses about the perspectives.

Chapter 2

State of The Art

Contents

2.1 Chapter Overview

This chapter describes background concepts and principles relevant to the thesis and some related work to our approach. The chapter is organized as follows:
Section 2.2 presents an overview of cloud computing and its concern concepts: Service delivery model, deployment model, and virtualization. This section also describes the configuration management of Virtual Machine Image (VMI) and the VMI provisioning process.
Sections 2.3 summarizes the fundamental concepts of automating software development, including Model-Driven Engineering, Software Product Lines and Feature Modeling.
Section 2.4 presents the feature modeling approach for VMI configuration management. In this section, we describe the fundamental concept of product line engineering, feature model and how to represent the VMI configurations as the Product Lines. In addition, we discuss on the state of the art of feature modeling for VMI configuration management and related work.
Section 2.5 discusses about the state of the art and related work on the deployment process of VMIs in cloud computing.
Finally, Section 2.6 summarizes the chapter.

2.2 Cloud Computing

2.2.1 An overview of cloud computing

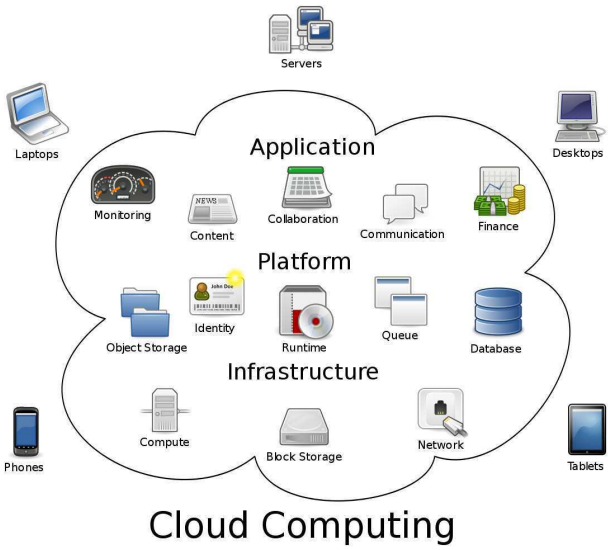


Figure 2.1: Cloud Computing

The term "Cloud Computing" is currently a hot and highly discussed topic in both technical, economic, and research world. It is used for describing what happens when applications and services are moved into the "Cloud". Actually, cloud computing is not so new, in some cases it may consider as a new form of computer systems, which are remotely time-shared computing resources and applications. However, more currently though, cloud computing refers to many different types of services and applications being delivered in the internet cloud, and the fact that, in many cases, the devices used to access these services and applications do not require any special platforms or infrastructures; see Figure 2.1. Many big companies within the IT industry (e.g., Microsoft, IBM, Google, Amazon,..) are joining to the development of cloud computing, and providing cloud computing services [7, 24, 41]. However, cloud computing definition remains unclear. Many people within the industrial and academic community have attempted to define what "Cloud Computing" really is, and what typical characteristics it presents. Armbrust *et al.* [7] define a cloud as the *"data center hardware and software that provide services"* and summarize the key characteristics of cloud computing as: *(1) the illusion of infinite computing resources; (2) the elimination of an up-front commitment by cloud users; and (3) the ability to pay for use as needed "* .
Buyya *et al.* [14] have defined cloud computing as follows: *"Cloud is a parallel and distributed computing system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level-agreemments (SLA) established through negotiation between the service provider and consumers "*

The National Institute of Standards and Technology (NIST)[1] proposes the following definition of cloud computing: *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* [32]. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

2.2.1.1 Essential characteristics of cloud computing

1. *On-demand self-service:* A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

2. *Broad network access:* Capabilities are available over the network and accessed through standard mechanisms that promote the use by different types of client platforms (e.g., mobile phones, tablets, laptops, and workstations).

3. *Resource pooling:* The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify the location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.

4. *Rapid elasticity:* Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly on demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

5. *Measured service:* Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate for the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

2.2.1.2 Service delivery models of cloud computing

Figure 2.3 shows the general model of cloud computing service with the roles of services and actors. The differences between the cloud computing services depend upon the capability of cloud providers, and they are related with the type of service offered, such as: (1) storage and computing capacity, (2) platform for own software development and testing, (3) online software applications. According to these differences, NIST has already proposed three main service delivery models of cloud computing services:

1. **Infrastructure as a Service (IaaS):** provides virtualized infrastructure to the user, such as storage, network, processing and other computing resources. Cloud users are

[1] http://www.nist.gov/

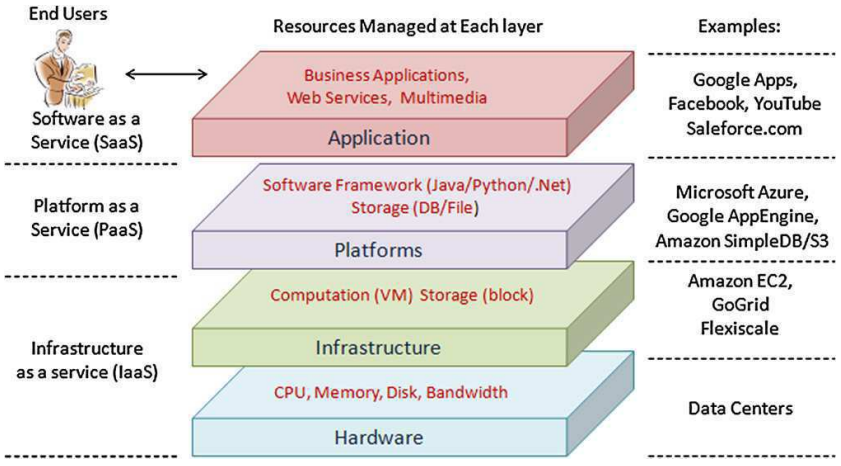


Figure 2.2: Service delivery models of cloud computing [25]

able to deploy and run software, which can include operating system and applications. It enables on-demand provisioning of servers with different choices of operating systems and software stacks. IaaS is considered as a bottom layer of cloud computing systems. For example: Amazon Web Services or IBM Smart Cloud mainly offers IaaS.

2. **Platform as a Service (PaaS):** offers development platforms where the developers can design, develop, deploy and test their applications and do not necessary need to know about the infrastructure (e.g., processors, memory, storage). There are several IT vendors have developed new PaaS systems such as Google App Engine[2], Microsoft Azure[3] and Amazon Web Service[4].

3. **Software as a Service (SaaS):** provides services or composition services to users. These services run on a cloud infrastructure and are accessible by various client devices through a client interface such as Web browsers (e.g. Web mail, Google Docs, etc.) or client applications (e.g., iTunes, Picasa, etc.).

2.2.1.3 Deployment models of cloud computing

Cloud computing has emerged from the combination of public computing utilities. However, depending upon the purpose of cloud services, and physical location or distribution of cloud systems, a cloud can be classified as a public, private, community or hybrid cloud [32, 7].

1. *Private cloud:* A cloud system is operated solely for a single organization. In other words, the proprietary network or the internal data center supplies hosted services to a certain group of people, not made available for general public users. For example, Microsoft Azure enables customers to build the foundation for a private cloud infrastructure using Windows Server and System Center family of products with the Dynamic Data.

[2]http://code.google.com/appengine
[3]http://www.microsoft.com/windowsazure
[4]http://www.amazon.com/aws

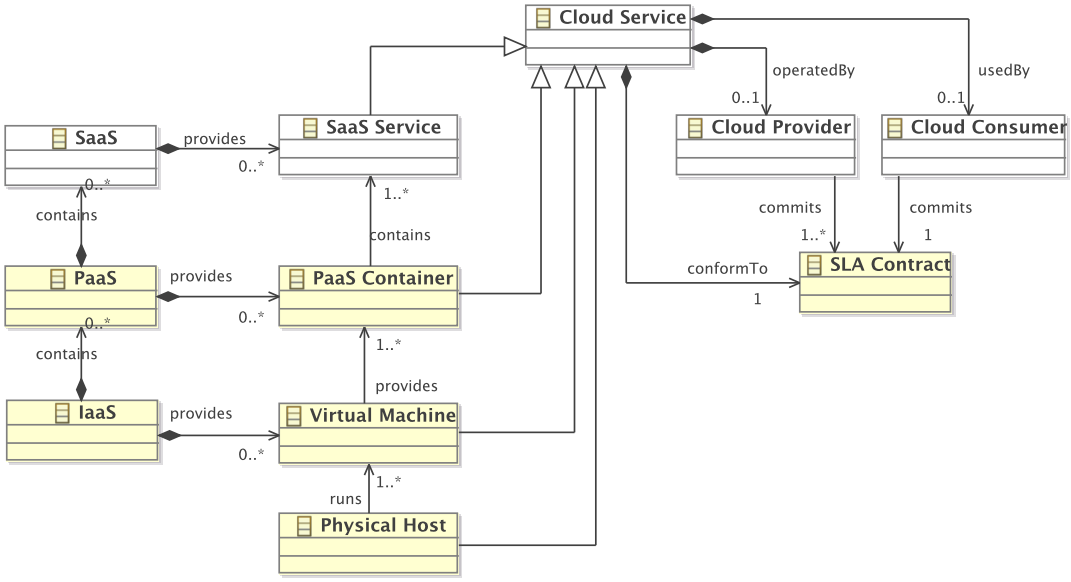


Figure 2.3: A cloud computing metamodel

2. *Public cloud:* A cloud service provider makes resources (e.g., applications and storage) available to the general public over the Internet on a pay-as-you-go basis. For example, the Amazon Elastic Compute Cloud (EC2) allows users to rent virtual computers on which they run their own applications. It provides network infrastructure, data centers and allows customers to pay only for what they use with no minimum fee.

3. *Community cloud:* The cloud system is shared by several organizations with common concerns (e.g., mission, security requirements, policy, and compliance considerations).

4. *Hybrid cloud:* The cloud infrastructure comprises 2 or more clouds (e.g., private, public, or community). In this infrastructure, an organization provides and manages some resources within its own cloud and has other resources provided externally from other clouds.

2.2.2 Virtualization technology in cloud computing

2.2.2.1 Virtualization

In cloud computing, virtualization is one of the key features to unlock values for a cloud system. The different customers with disparate requirements require a flexible managing resource. Virtualization can provide significant benefits for a computing system, including increased utilization, energy saving, rapid deployment, improved maintenance capability, isolation, and encapsulation. Moreover, virtualization enables applications to migrate from one server to another while they are still running, without downtime, providing flexible workload management, and high availability during planned maintenance or unplanned events [6].
The idea of virtualizing the computer 's resources such as processors, memory, storage devices

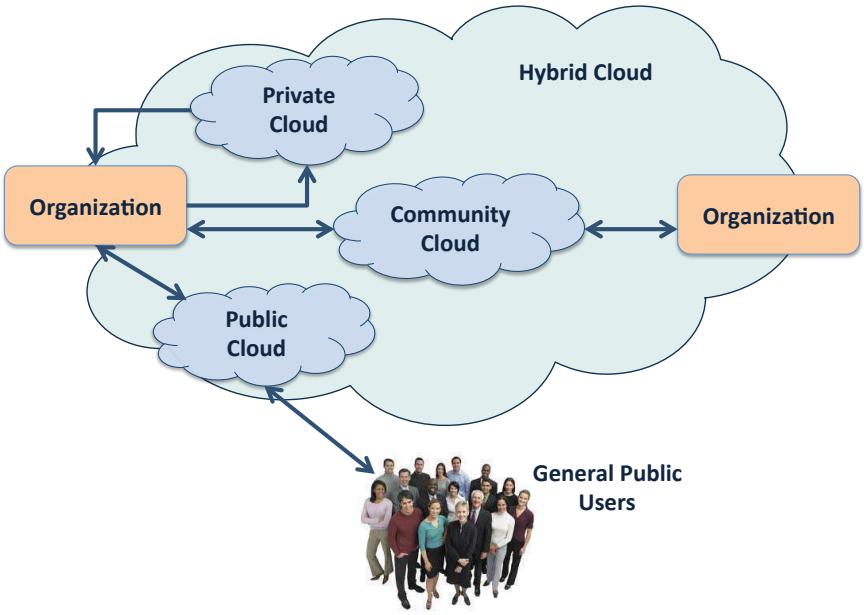


Figure 2.4: Deployment models of cloud computing

have been established since last decades, aiming for improving sharing and utilization of computer systems [22]. Virtualized resources (e.g., CPU, memory, etc.) scale with certain flexibility and create a virtual infrastructure. Virtualization allows running multiple operating systems and software on a single physical platform. Figure 2.5 shows a server using virtualization technology for hosting different virtual machine (VM). Each of these VMs running a distinct operating system and software stack. A hypervisor is a set of virtual platform interfaces; it is a component that is responsible to manage virtual machines. It supports a mediate access to the physical hardware available for each guest operating system of virtual machines. There is some notable hypevisor platform, such as Xen[5], KVM[6], VMWare ESXI[7], or VirtualBox[8].

- **Xen:** Xen is an open-source hypervisor. It has pioneered the para-virtualization concept, on which the guest operating system can interact with the hypervisor, thus significantly improving performance [13]. Xen is used as the basis for a number of different commercial and open source applications, such as: server virtualization, Infrastructure as a Service (IaaS), desktop virtualization, security application, embedded and hardware appliances. Xen enables users to increase server utilization, consolidate server farms, reduce complexity, and decrease total cost of ownership. It currently forms the base of commercial hypervisors of many cloud vendors (e.g, Citrix XenServer[9], Oracle VM[10]).

[5]http://www.xen.org/
[6]http://www.linux-kvm.org/
[7]http://www.vmware.com/
[8]https://www.virtualbox.org/
[9]http://www.citrix.com/products/xenserver/overview.html
[10]http://www.oracle.com/us/technologies/virtualization/oraclevm/overview/index.html

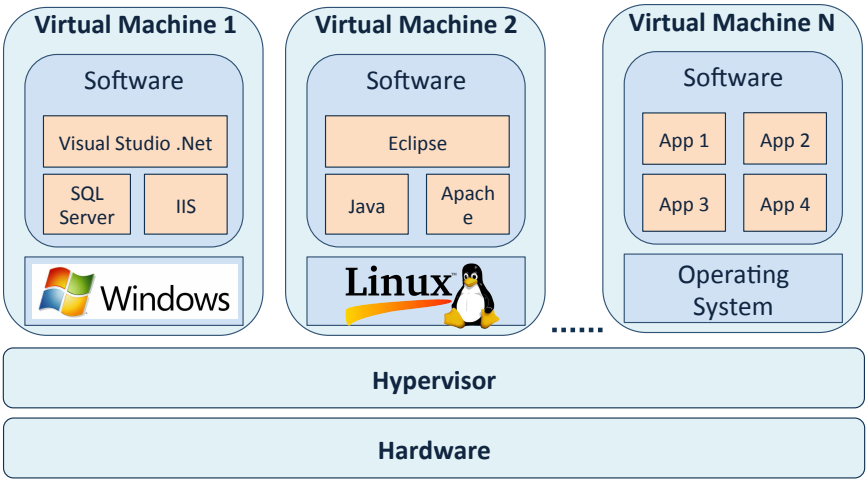


Figure 2.5: Virtual machines running on a virtualized server

- **KVM:** The kernel-based virtual machine (KVM) is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module, kvm.ko, that provides the core virtualization infrastructure and a processor specific module, kvm-intel.ko or kvm-amd.ko. Using KVM, one can run multiple virtual machines running unmodified Linux or Windows images. Each virtual machine has its own private virtualized hardware: a network card, disk, graphics adapter, etc.

- **VMWare ESXI**: VMWare is one of the leader on the virtualization market. VMWare ESXI hypervisor is a commercial product of VMWare. It provides advanded virtualization techniques of processor, memory, and I/O. It is a bare metal embedded hypervisors, which means that they run directly on server hardware and do not require the installation of an additional underlying operating system. This virtualization software creates and runs its own kernel, which is run after a Linux kernel bootstrap the hardware. The resulting service is a microkernel, which has three interfaces: Hardware, Guest system, Console operating system (service console).

- **VirtualBox:** VirtualBox or Oracle VM VirtualBox is a cross-platform virtualization software for x86-based systems. "Cross-platform" means that it installs on Windows, Linux, Mac OS X and Solaris x86 computers. VirtualBox is installed on an existing host operating system as an application; this host application allows additional guest operating systems, each known as a Guest OS, to be loaded and run, each one with its own virtual environment.

2.2.2.2 Virtual Machine and Virtual Machine Image (Virtual Appliance)

To deliver highly available and flexible services by using virtualization technology, Virtual Machines (VMs) are used as a standard for object deployment in the cloud. VMs decouple the computing infrastructure from the physical infrastructure. In addition, VMs allow the

customization of the platform to fit the needs of the end-user [6]. For example, in the Amazon Elastic Compute Cloud (EC2), the customer selects his/her preferred VM image (VMI) from a list of various versions of Linux and Windows servers configured with different web servers and databases. The difference between a virtual machine and a virtual machine image (also called a virtual appliance) is:

- Virtual Machine (VM): A VM is a software implementation of a machine (i.e., a computer) that executes programs like a physical machine. It offers many advantages over physical PCs and can encapsulate an entire PC environment, including the OS, applications and all data inside a single file. However, users must still configure the virtual hardware, guest operating system and guest application before putting a virtual machine into operation.
- Virtual Machine Image (VMI): is an application combined with the environment needed to run it (operating system, libraries, compilers, databases, application containers, etc.) [13]. Virtual machine images differ from virtual machines in that they are delivered to customers as preconfigured solutions, which helps to simplify the deployment for customers by eliminating the need for manual configuration of the virtual machines and operating systems used to run the image.

2.2.3 Requesting and provisioning processes of cloud services

2.2.3.1 A request for service

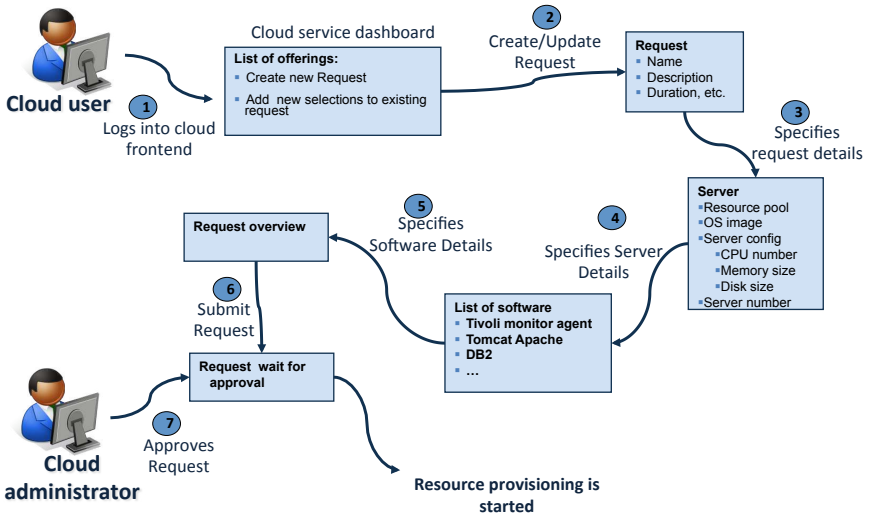


Figure 2.6: A scenario of a request for cloud service

Consumers of cloud computing expect to use on-demand services. Therefore, cloud providers have to support self-service access so that customers can request, customize and use services without intervention of human operators [32]. In thesis, we consider the requesting and provisioning processes of cloud services - virtual machine images for IaaS environments.
Figure 2.6 presents a scenario of requesting a VMI in cloud computing. Cloud user accesses

the cloud system by login into the cloud frontend. Basing on offering service from the cloud providers, he will create a request or modify an existing one. In the request, he can define detailed requirements on the service, such as information of server (e.g., CPU, memory, storage, etc.) and software stacks. Cloud administrators are responsible to review and process the requests. When the request is approved, then the cloud service with corresponding resources is provisioned.

2.2.3.2 A service provisioning process

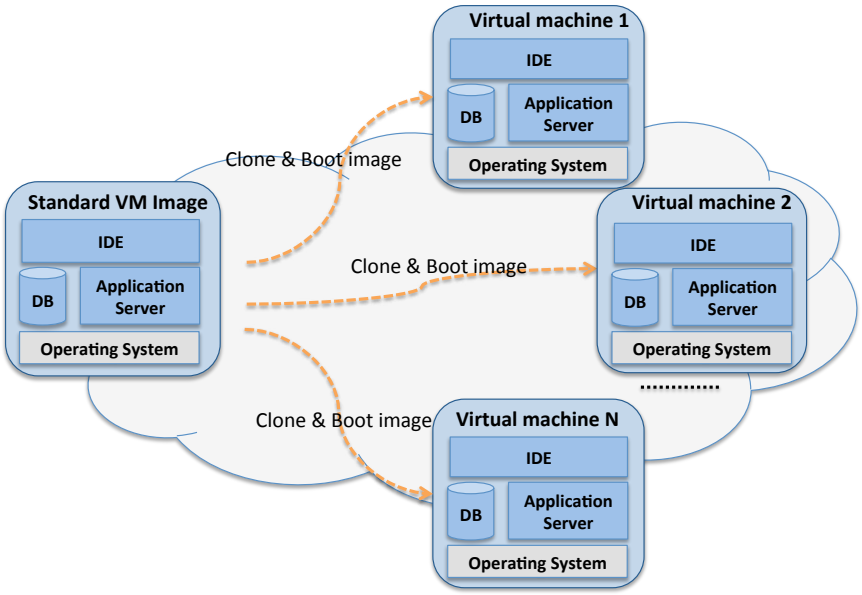


Figure 2.7: A traditional approach for VMI provisioning in cloud computing

In the above scenario, cloud providers created different VMIs with distinct software stacks installed inside. After reviewing the request from a cloud user, cloud administrators select an appropriate existing VMI (template VMIs or Golden Images) and deploy it on cloud nodes by using the virtual infrastructure manager. If there is no existing VMI that matches the requirements, then a new one is created and configured to match the request. It can be generated by modifying from the closest-fit existing VMI or from scratch. This method is called traditional approach of VMI provisioning in cloud comptuting [2, 1], see Figure 2.7. Once created, the template VMI is not executed. Instead, it is the source of copies or clones that are replicated to create one or many instances of the template VMI.

2.2.3.3 Discussion

The template virtual image contains all possible software and can be cloned and booted as many times as the deployment model requires. Therefore, it exposes the disadvantages to the approach. In the context of an enterprise data center using file systems or storage systems with built-in, space-efficient cloning mechanisms, the template VMI can be copied almost instantly because actual data blocks are replicated only when they are changed by a running instance.

However, in cloud computing environments, cloud providers must maintain the ability to transfer a VMI to any storage system or deploy to cloud nodes regardless of the location of the template VMI. While adapting for various requirements from distinct customer, the template VMIs may contain many kinds of software stacks inside. It makes the size of template VMIs bigger than needed. So in a cloud environment, actual copies of the template VMI must be created and potentially moved across the network. As cloud data physics indicate, this takes time and bandwidth, limiting the responsiveness of the approach [48, 3].

2.3 Model-Driven Engineering

2.3.1 Model, Metamodel and Modeling, Metamodeling

- **What is a model and metamodel?**
  "We use models when we think about problems, and when we talk to each other, and when we construct mechanisms, and when we try to understand phenomena, and when we teach. In short, we use models all the time" [30].
  "A model is a simplified representation of an aspect of the world for a specific purpose. Complex systems typically give rise to more than one model because many aspects are to be handled." [26]
  A model is usually considered as *an abstraction, a description, or a specification of (some aspect of) a system* [19, 21].
  While a model is an abstraction of phenomena in the real world; a metamodel is yet another abstraction, highlighting properties of the model itself. A model conforms to its metamodel in the way that a computer program conforms to the grammar of the programming language in which it is written.

- **What is modeling and metamodeling?**

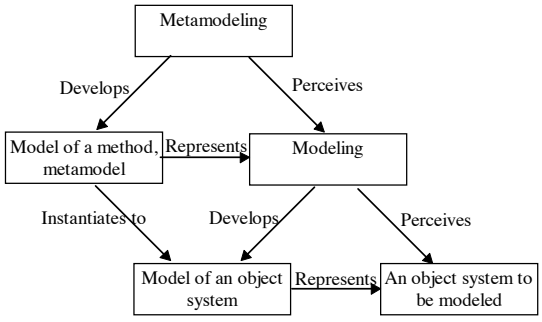


Figure 2.8: Relationships between Model, Metamodel, Modeling and Metamodeling [46]

Modeling not only represents a solution at a higher abstraction level than code, but also is a specificity of engineering that engineers build models of artefacts that usually do not exist yet [26]. It is indeed one of the standards of any scientific activity along with validating models with respect to experiments carried out in the real world.

Metalmodeling is the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for modeling a predefined class of problems. As its name implies, this concept applies the notions of meta- and modeling. In short, "Metamodeling" is the construction of a collection of "concepts" (things, terms, etc.) within a certain domain. The relationships between model, metamodel, modeling and metamodeling are illustrated in Figure 2.8.

2.3.2 Model-Driven Engineering and Model-Driven Architecture

- **Model-Driven Engineering**
  Model-Driven Engineering (MDE) refers to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle. Models are considered as first class entities, they are better than implementing code for describing about software system and its parts for the stakeholders. MDE can be applied to software, system, and data engineering. It offers a promising approach to address the inability of third generation languages to reduce the complexity of platforms and express domain concepts effectively [42]. Applying MDE approach for the deployment process helps to encapsulate the deployment and management of a cloud system into a series of procedural operations. Schmidt *et al.* [42] summarized that a promising approach to address the system complexity is to develop MDE technologies that combine 1) domain-specific modeling languages (DSML) whose type systems formalize the application structure, behavior, and requirements within particular domains; and 2) transformation engines and generators that analyze certain aspects of models and then synthesize various types of artifacts, such as source code or alternative model representations.

- **Model-Driven Architecture**
  Model-Driven Architecture (MDA) is an MDE approach proposed by the Object Management Group (OMG)[11]. As defined by the OMG, MDA encourages the efficient use of system models in the software-development process, and it supports reuse of best practices when creating families of systems. MDA is a way to organize and manage enterprise architectures supported by automated tools and services for both defining the models and facilitating transformations between different model types [12].
  The main goal of MDA is to support engineers to model applications, or systems independently of specific platforms. MDA defines system functionality using a platform-independent model (PIM) using an appropriate domain-specific language (DSL). Then, the PIM model is translated into one or more platform-specific model (PSM) that computers can run. This requires mappings and transformations. One of the particular importance to model-driven architecture are the notions of metamodel and model transformation. Metamodels are defined at the OMG using the MOF (Meta Object Facility) standard. A specific standard language for model transformation called QVT has been defined by OMG. The OMG has also defined a model interchange mechanism based on XML called XMI.

[11] http://www.omg.org/mda/

2.3.3 Domain-Specific Modeling

Domain-specific modeling (DSM) is a specific software engineering method for designing and developing systems based on the systematic use of a domain-specific modeling language (DSML). DSMLs tend to support higher-level abstractions than general-purpose modeling languages, therefore they require less effort and fewer low-level details to specify a given system.
DSMLs are described using metamodels, which define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts. Developers use DSMLs to build applications using elements of the type system captured by metamodels and express design intent declaratively rather than imperatively [42].

2.3.4 Model@Runtime

G. Blair and R.France [11] define: *" A model@run.time is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals within the system from a problem space perspective "* .
The model@runtime techniques provide a high abstraction level for solving the adaptation issues of the running system by reasoning with relevant abstractions (as models) of the system [38]. It aims to handle the complexity of dynamic adaptation of the systems at runtime [39]. When changes that apply to the running system occur in a new model (a target model), the model is checked and validated to ensure the consistency of the system configuration. Then, it is compared to the current model which is the representation of the running system. This comparison finds the changes and produces an adaptation model which represents a mechanism to reach a target model from the current one. It supports the roll-back mechanism for any actions failed to ensure the consistency to the system.

2.4 Feature Modeling for VMI

2.4.1 Software Product Lines and Feature Modeling

2.4.1.1 Software Product Lines

In software engineering, the traditional focus is to develop individual software systems, i.e, one software system at a time. The result obtained is a single software product. In contrast, *Software Product Line (SPL)* engineering focuses to develop multiple similar software systems from the common assets [18, 40]. Clements *et al.* [18] define SPL as: *"a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."*. It captures "commonality" and "variability" between a set of software products in the same domain. Commonality refers to elements that are common to all products while variability refers to elements that may vary from a product to another one. Using SPL helps to improve productivity and reduce realization times by gathering the anlysis, design, and implementation activities of a family of products [49, 27].

The process of SPL engineering [19, 40] includes two steps: (1) *Domain Engineering* focuses on core assets development; (2) *Application Engineering* addresses the development of the final products based on core assets and user requirements. This process is illustrated in Figure 2.9.

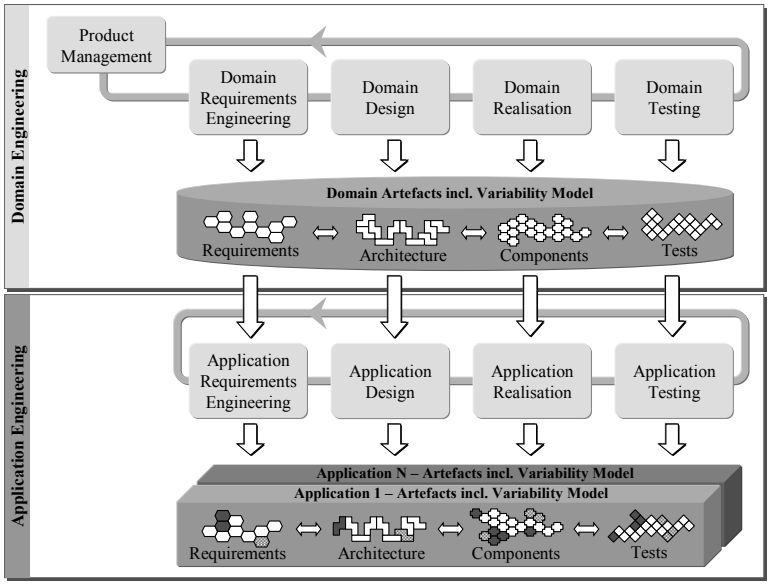


Figure 2.9: Two engineering process of SPL Engineering [40]

- **Domain Engineering** *(development for reuse)*: includes the collecting, organizing and storing past experiences in building systems in the form of reusable assets in particular domain; and providing means for reusing these assets when building new systems [19]. It starts with *domain analysis* phase to capture, analyze and organize information as a model (define the commonalities and variability of products). Then, *domain design* is responsible for establishing the product line architecture in terms of software components, and the last phase is responsible for implementation [27].
- **Application Engineering** *(development with reuse)*: also known as product derivation. This process consists of developing a new product, reusing the reusable assets from *domain engineering*, and adapting the new product to specific requirements. Therefore, the new product is built from existing common and variable parts of the SPL [49].

There are some major concepts of SPL that help the representation of product lines: *the commonality*, *the variability*, *the variability dependencies*, and *the variability constraints*.

1. **The commonality**
   The commonality of product lines allows to specify the same features between products of a certain group (*a product line* or a product family). It provides the *common platform* for building a set of similar products. The common platform provides the structure of basic components for constructing the core characteristic of a product.

2. **The variability**
   The variability of product lines provides the pre-definition of what components could be assembled into a product. It supports to define exactly the places where the products can differ from the others. A variability model is the modeling of the variability of objects in the real world.

In the real world, the term *variability subject* means a variable item of the real world or variable property such an item (e.g. *Database*, *Operating System*, or *Web Server*, etc.), and *a variability object* specifies a particular instance of a *variability subject* (e.g. *MySQL*, *Windows 7*, or *Apache Tomcat*, etc.).
In the context of product line engineering, two terms, *a variation point* and *a variant*, are used for representing the variability of the real world 's problems in the variability models. A *variation point* is a representation of a variability subject within domain artefacts enriched by contextual information; and a *variant* is a representation of a variability object within domain artefacts.

3. **The variability dependencies**
   The variability dependencies specify the relation between variation points to their variants. The variability dependencies can be classified as two types: *Optional* and *Mandatory*.

   - *An optional dependency:* states that a variant can be selected or not for being a part of the product.
   - *A mandatory dependency:* states that a variant must be selected when the variation point is part of the product.

4. **The variability constraints**
   The variability constraints describe the relationship between a variant and another variant or a variation point. It can be a *requires* or *excludes* constraints.

   - A requires constraint: defines that the selection of a variant requires the selection of another variant.
   - An excludes constraint: defines that the selection of a variant excludes the selection of another variant.

2.4.1.2 Feature Modeling

Feature modeling is a variability modeling technique, originally introduced as a part of Feature-Oriented Domain Analysis (FODA) by Kang [28]. Then, it has been developed and used in automotive industry, telecom or embedded systems. By using this technique, engineers can capture, analyse and manage the commonalities and variabilities of product families. In software development, feature models [28] represent all products of a Software Product Line (SPL). They are currently the standards for representing variability. A feature model has a tree structure, with features forming the nodes of the tree and groups of features representing feature variability. It defines a set of valid feature configurations. The validity of the feature model and its derived configurations relies on the semantics of the feature model. For example, the model must follow a set of rules, such as:

- All its *mandatory* child features must also be contained;
- Any number of *optional* child features can be included;
- Exactly one feature must be selected from an *alternative* group;
- At least one feature must be selected from an *or* group.

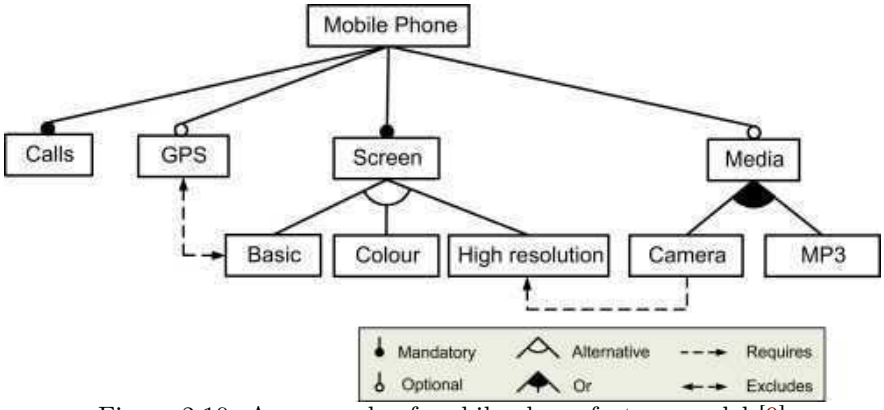


Figure 2.10: An example of mobile phone feature model [9]

In addition, the semantics of a feature model also supports the specification and reasoning about the commonality and variability of a product line [4].

Feature models support two cross-tree constraints: *Requires*, and *Excludes*; and four types of feature groups: *Mandatory*; *Optional*; *Alternative*; and *Or*.. For example: Given two features, f_a and f_b: if f_a requires f_b, then the selection of f_a implies the selection of f_b; if f_a excludes f_b, then the selection of f_a prevents the selection of f_b. Many approaches and tools were proposed to automate analysis of feature models for checking and validating the correctness of feature models [45, 10, 36].

2.4.1.3 Feature modeling tools

There are many tool support to the variability management by using the feature model, such as FeatureIDE [12] [17], pure::Variant [13], FAMILIAR [14][5], etc. Because our work does not focus on feature model development, we used feature model as a tool for managing the VMI configuration options. Therefore, in this thesis, we re-used and extended the SPLAR (A Software Product Line Automated Reasoning) engine for the reasoning process of feature modeling. It is an open-source library developed by M.Mendoca in his PhD work. This library offers SAT and BDD-based components to reason on and to configure feature models [37, 34]. SPLAR allows us to validate the feature selections, and to generate the valid configurations from selected features and dependencies. Detail of how we use and extend the SPLAR engine will be described in Chapter 3 and Chapter 5.

2.4.2 VMI Configurations as Product Lines

2.4.2.1 VMI Configurations

A virtual machine (VM) is a software implementation of a machine (i.e., a computer) that executes programs like a physical machine [22, 44]. It offers many advantages over physical PCs and can encapsulate an entire PC environment, including the OS, applications and all data, inside

[12] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/
[13] http://www.pure-systems.com/pure_variants.49.0.html
[14] http://familiar-project.github.com/

a single file. However, users must still configure the virtual hardware, guest operating system and guest application before putting a virtual machine into operation. A virtual machine image (VMI) is an application combined with the environment needed to run it (operating system, libraries, compilers, databases, application containers, etc.). VMIs are delivered to customers as pre-configured solutions, that include operating systems with pre-built, pre-configured and ready-to-run applications. This approach simplifies the deployment for customers by eliminating the need for manual configuration of the virtual machines and operating systems used to run the image.
Because there is no single image that can fit all possible combinations of requirements, cloud providers must prepare different VMI configurations. For example, some users want to use Windows operating system, while others like Linux; some people need a database while some other need integrated development environment, etc. It leads to wasted time, and cost. The

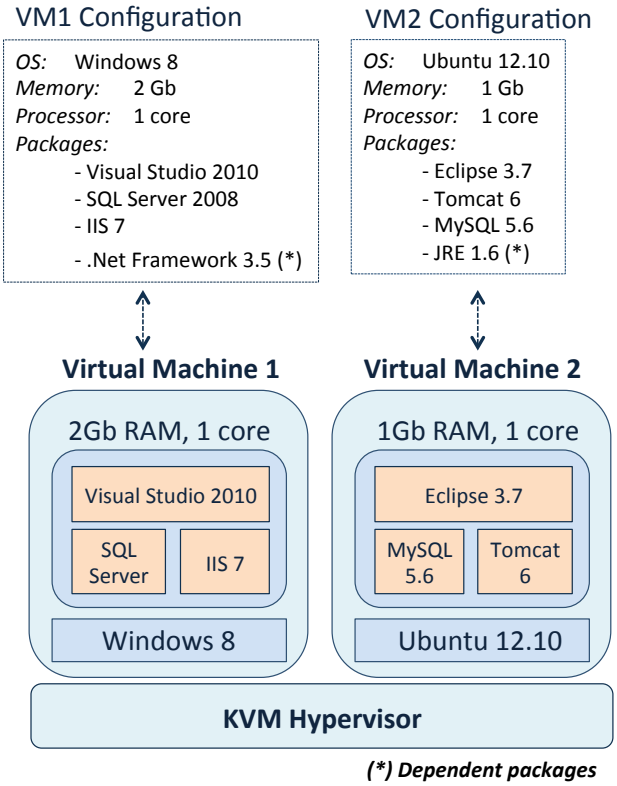


Figure 2.11: An example of two virtual machines run on KVM hypervisor

configuration of a VMI represents the basic information of an image (e.g. RAM, Processor, etc.) and packed software components. Depending on the requirements from users, cloud providers create the appropriate images which contain the requested software and the dependencies. The VMI configurations must handle the dependent packages, and satisfy the constraints of the requested software. Figures 2.11 shows the example of two virtual machines with different configurations run on the KVM hypervisor. The first VMI configuration includes *IIS 7, Visual Studio 2010, MS SQL 2008,* and *.Net Framework 3.5* run on *Windows 8* operating system

with *2Gb RAM* and *1 core* processor. While the second VMI configuration contains *Tomcat 6, Eclipse 3.7, MySQL 6.5, JRE 1.6* and Linux *Ubuntu 12.10* operating system with *1Gb RAM, 1 core* processor. *.Net Framework 3.5, JRE 1.6* are dependent packages that are required by *IIS 7, SQL Server 2008* and *Eclipse 3.7, Tomcat 6* respectively. In addition, with 2Gb memory, the VM1 configuration also satisfies the constraint *"Visual Studio 2010 needs at least 1Gb RAM"*.

2.4.2.2 VMI Configuration Commonality

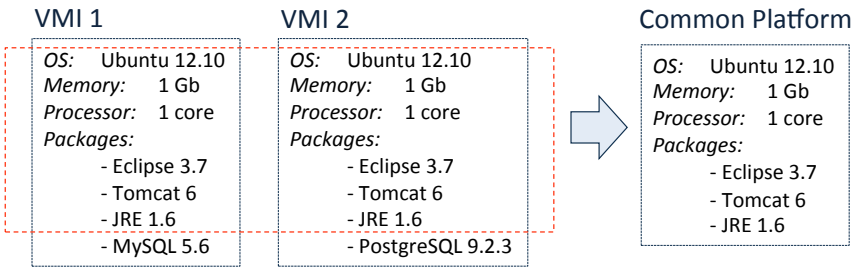


Figure 2.12: An example of the VMI Configuration commonality

The commonality of VMIs used to specify the same features between images of a certain group of VMIs (*a VMI product line* or *a VMI product family*) [20]. It supports to share the *common platform* for building and customizing a line of VMIs that have similar characteristics. The *common platform* provides a structure of base components determining the major characteristics of a virtual machine, such as VMIs for hosting database servers or web application servers. By determining the common platforms of the existing VMI configurations and the expected VMI configurations, cloud providers can easily find the best-fit VMIs for re-using and customizing.
The use of the common platform for different VMIs typically leads to a reduction in the production cost, time and error-prone of a particular VMI, and it improves the performance of the provisioning process. Figure 2.12 shows an example of the similarities between two VMIs and the common platform of these images. Therefore, it is easy to generate the VMI 2 by customizing the VMI 1 (e.g. remove *MySQL 5.6* and install *PostgreSQL 9.2.3*) or vice versa.

2.4.2.3 VMI Configuration Variability

Different VMI configurations of the same product line may contain different packed software components. The VMI configurations are designed in a way that allows the cloud providers fulfill the different user 's requirements, such as the number of processors, memory, etc. Such flexibility comes with many constraints. For example, if a user wants to use *Visual Studio with .Net Framework* for programming, then the operating system of the VMI must be *Windows*, and the selection of other components, such as databases that run on Linux are restricted, therefore they are disabled when the *Windows* operating system is selected. The flexibility of VMI configurations described here is called *variability* in the context of *software product line* engineering. The variability of VMI configurations provides the pre-definition of what possible software shall be installed into a VMI. In addition, it supports to define exactly the places where the VMI can differ from others, so that they can have as much in common as possible.

Figure 2.13 shows an example of the variability of database systems for a VMI, as well as the

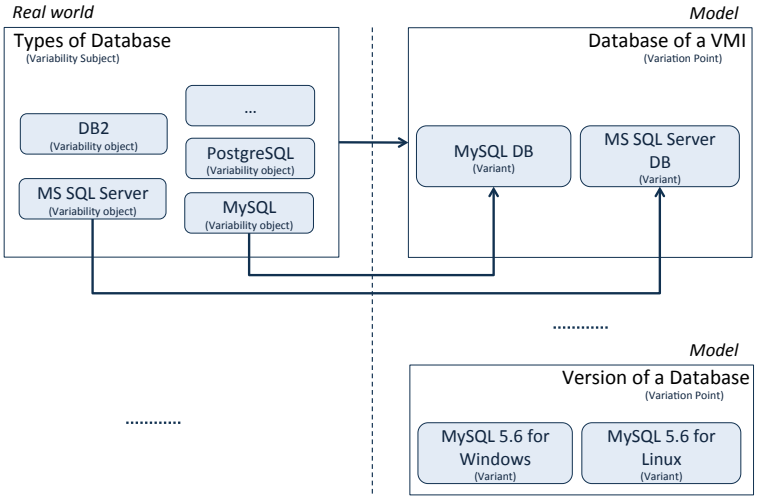


Figure 2.13: An example of the VMI configuration variability in the real world and in the model of the real world

relation between variability in the real world and in the model of the real world. From example, there are many types of database systems, when the cloud provider installs a database system into a VMI, it could be *MySQL*, or *SQL Server*, etc., and each type of them is also classified into different versions. The left side of Figure 2.13 presents the database system variabilities in the real world with *variability subjects* and their *variability objects*. The right side the figure presents the database system variabilities in the model of the real world with *variant points* {*Types of Database*, etc.} and their *variants* {*DB2, MS SQL Server, MySQL, PostgreSQL*}. Mapping to the context of software product line engineering, the terms *variability subject, variability object, variant point,* and *variant* can be applied as follows [40]:

- *The variability subjects* are *Operating Systems*, *Databases*, or *Programming Languages*, etc.
- *The variability objects* are *Ubuntu 12.10*, *Windows 7*, etc., or *MySQL*, *SQL Server*, etc.
- *The variant points* are *RAM of a VMI*, *Number of processor*, etc.
- *The variants* are *1 GB*, *1 core*, etc.

2.4.2.4 VMI Variability Dependencies and Constraints

A VMI configuration can be specified by the selection of software options (*variants*) in the VMI base model. A legal product configuration of a VMI contains all selected software options and their dependencies which satisfy the corresponding constraints. For instance, a configuration of a VMI that contains *Ubuntu 12.04 TS*, *JRE 1.6 Linux*, and *Eclipse 3.7 Linux* is a valid configuration. However, another configuration *Ubuntu 12.04 TS*, *Java Runtime*, and *VisualStudio 2010* is an illegal configuration since features *Ubuntu 12.04 TS* and *VisualStudio 2010*

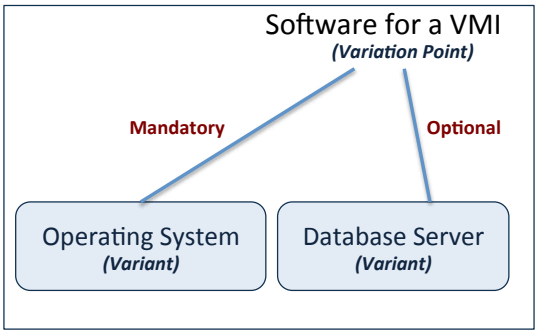
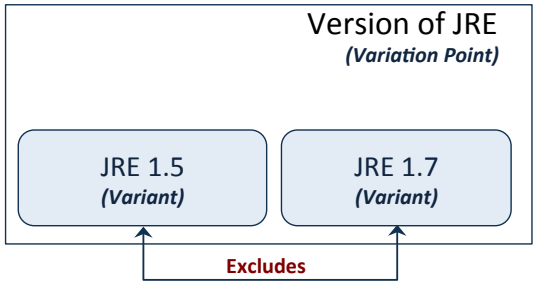


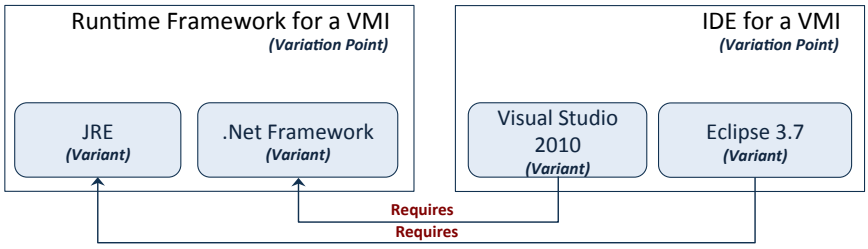
Figure 2.14: An example of variability dependency

are mutually-exclusive but appear together in a configuration. Figure 2.14 represents the variation point *"Software for a VMI"* and two variants *Operating System*, and *Database Server*. It specifies that *Operating System* must be a part of a virtual machine image while *Database Server* can (but does not need to) be installed in the image. .

Variability Constraint



(a) A variant excludes another variant



(b) A variant requires another variant

Figure 2.15: An example of variability constraints

Figure 2.15 presents an example of variability constraints. The *excludes* constraint in Figure 2.15a describes that if the variant *JRE 1.5* is selected, then the selection of variant *JRE*

1.7 is restricted, and vice versa. Figure 2.15b shows that if *Eclipse 3.7* or *Visual Studio 2010* is selected, then the variant *JRE* or *.Net Framework* must be selected respectively.

2.4.3 The Configuration Management of VMIs

In the investigation the state of the art, we found that there are some research efforts use feature models to capture configuration options of complex systems and helps to simplify the selection of configuration options. The use of feature models and cross cutting constraints for managing the configuration can reduce requirement elicitation errors and support automated choice propagation. Some other researches used the model of probabilistic analysis for generating different virtual image configurations that are contain a set of frequently used software packages. This approach helps to minimize the transformation time from an existing virtual appliance to a new one that fits the request. Especially, Dougherty et al. [20] present a technique to minimize

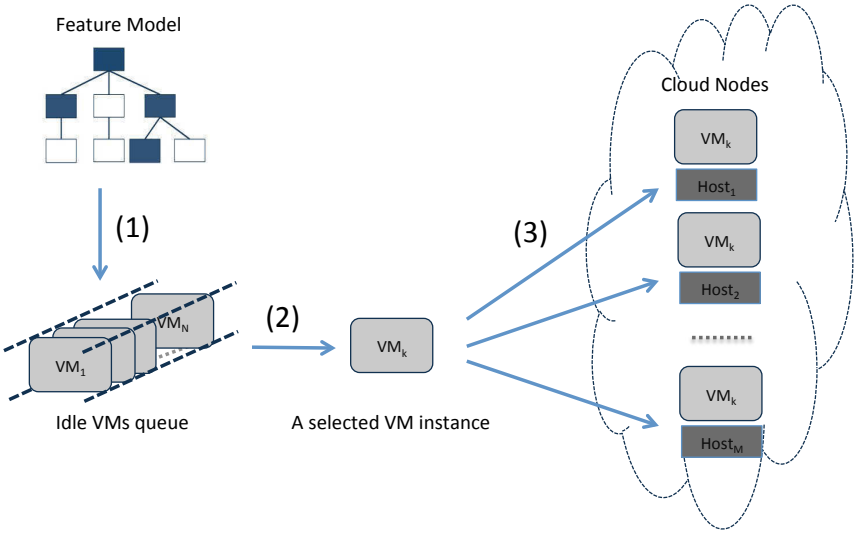


Figure 2.16: Feature model for VMs auto-scaling in Dougherty 's approach

the number of idle VMs in an auto-scaling queue. It helps to reduce the energy consumption and the operating cost and satisfies response time constraints. Their work defines a method to represent VMI configuration options by using feature models with constraints in the form of Constraint Solving Problems (CSP). It uses an auto-scaling queue to store created images in idle status. This leads to an improvement of response time when the request matches the available image in the queue. The overview of this approach is illustrated in Figure 2.16, where the feature model used to manage configuration options, and the idle VMs queue used to host pre-booted VMIs to provides faster response time. This approach is organized in three stages: (1) Transforming the VM configuration from the user 's selections on the feature model into a CSP for querying a suitable idle VM in the queue; (2) Selecting an idle VM from the queue that is exactly matches the user 's selections or modifying an idle VM that is closest to the requirements; (3) Deploying the selected idle VM to cloud nodes.
When a cloud user requests a virtual machine, the cloud provider analyzes the requirements

and generates a solution (a suitable configuration) by selecting the configuration options from the feature model, and then transforms this configuration into a CSP where a solution is a set of valid configurations for the VM instances in the auto-scaling queue. The valid configurations here is the configuration that exactly matches the target configuration, or the best-fit one that can be used for modifying while ensuring the minimization the energy consumption, cost of maintaining the auto-scaling queue and satisfying time constraints.

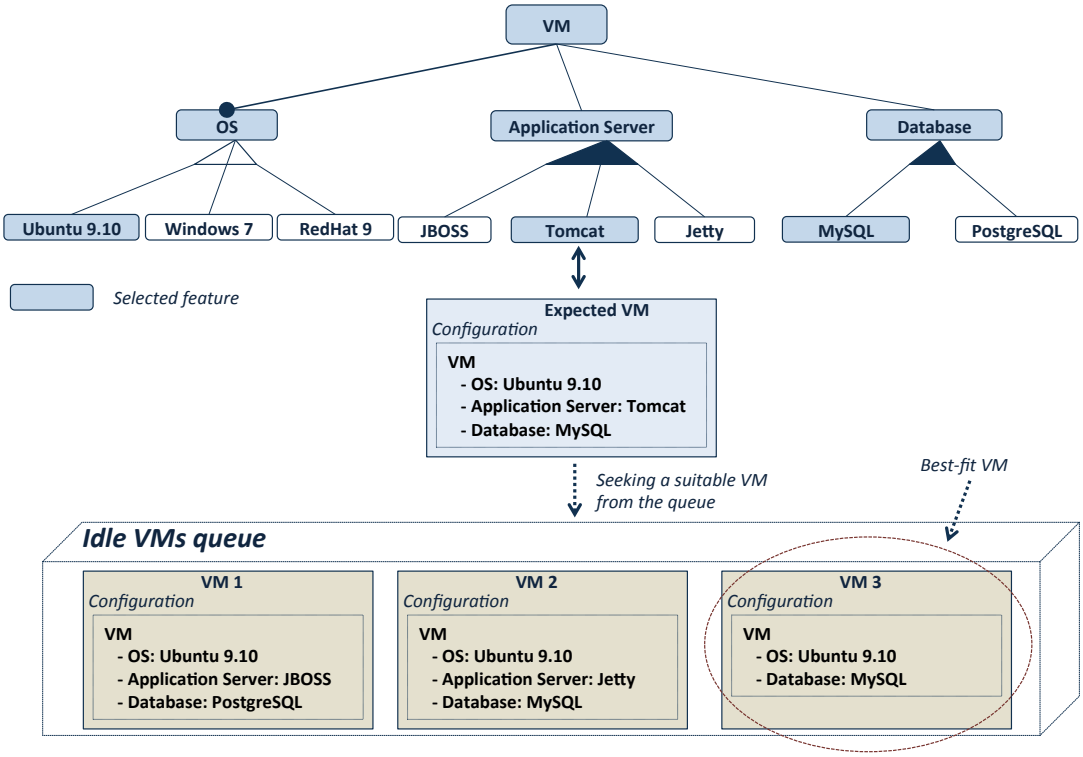


Figure 2.17: An example of VM configuration selections in Dougherty 's approach

Figure 2.17 is an example of feature selections from the feature model in Dougherty 's approach. The expected VM configuration is derived by these selections, and then matched to the idle VMs in the queue. It contains the following features: {*VM, OS, Ubuntu 9.10, Application Server, Tomcat, Database, MySQL}*. It means the expected VM will include a Linux *Ubuntu 9.10* operating system, a *Tomcat* application server, and a *MySQL* database server. However, in the auto-scaling queue with pre-booted VMs, there are only three idle VMs: **VM 1**{*VM, OS, Ubuntu 9.10, Application Server, JBOSS, Database, PostgreSQL*}; **VM 2**{*VM, OS, Ubuntu 9.10, Application Server, Jetty, Database, MySQL*}; and **VM 3**{*VM, OS, Ubuntu 9.10, Database, MySQL*}, and there is no one that exactly matches the expected VM configuration. Therefore, by using CSP objective functions, the cloud provider can select the best-fit VM whose modification minimizes the power consumption and operating costs of the queue. In the above example, the modification of idle VMs in the queue to be the expected VM is described in Table 2.1. Because every software take time for both installation and uninstallation,

Virtual Machine	Modifying steps	Estimation Time of Modifying
VM 1	- uninstall JBOSS - uninstall PostgreSQL - install Tomcat	Time = uTime(JBOSS) + uTime(PostgreSQL) + iTime(Tomcat)
VM 2	- uninstall Jetty - install Tomcat	Time = uTime(Jetty) + iTime(Tomcat)
VM 3	- install Tomcat	Time = iTime(Tomcat)

iTime(X): installation time of software X
uTime(X): uninstallation time of software X

Table 2.1: Example of the modifying existing VMs in the queue to fits the requirement

therefore the adjustment of virtual machine **VM 1** takes longer time than **VM 2**, and **VM 2** needs more time than **VM 3**. Therefore, the virtual machine **VM 3**{*VM, OS, Ubuntu 9.10, Database, MySQL*}is the best-fit VM in the queue because it takes shortest time for modifying and cloud provider just needs to install the *Tomcat* application server instead of modifying VM 2 or VM 1.

Two approaches have been introduced to solve optimization requirements of the creation of a new VMI based on the existent VMIs (Dougherty *et al.*) or from the probabilistic analysis of virtual appliance frequently used (T. Zhang *et al.*) to minimize the power consumption, operating costs of auto-scaling queue and transformation time of an existing VM to an expected VM, but they still have certain limitations:

- The probabilistic analysis of the virtual appliance frequently used in T. Zhang 's approach can meet the common requirements from the users with many similar interests as well as the habit of using the software (e.g. operating system, programming language, database, etc.) but lacks the flexibility to provide services on demand when those cloud users are diverse and require the use of services that are also different.

- In the approach of Dougherty *et al.*, the selections of users on the feature model should be concrete even when users do not care much about the details of the selection. Therefore, the search for the optimal solution will be difficult when the user requests the generic requirements (e.g. " - I need a database " instead of "- I want to use MySQL database", etc.). In addition, the maintenance of the auto-scaling queue with the idle VMs needs a certain amount of resources. In this case, user 's requirements do not fit any idle VM in the queue, thus the expected VM must be built from scratch, and leading to resources available in the queue that are not utilized.

- In both approaches, the composition of virtual images occurs at design time and at the administrator side, before the system copies and deploys them into cloud nodes. This makes it difficult to synchronize the maintenance and modification of the running images as needed when the amount of running cloud nodes is large. For example, upgrading the software version, or installing a new software package on the running virtual machines is going to be extremely costly.

2.5 The Deployment Process of VMIs

For the deployment process of virtual images in cloud computing environment, Konstantinou *et al.* [29] describe a model-driven engineering approach for virtual image deployment in virtualized environments. They focus on reusable virtual images and their composition. The authors introduce the concept of virtual solution models. This concept defines the solution as a composition of multiple configurable virtual images. The virtual solution model is an abstract deployment plan and it is platform-independent. According to the specific cloud platform, the model can be transformed into an executable deployment plan [23]. Chieu *et al.* [15, 16] and Arnold *et al.* [8] propose the use of virtual image templates. Their approaches describe a provisioning system that provide pre-installed virtual images according to the deployment scenario. M. Sethi *et al.* [43] present an approach for automated modification of dependency configuration in SOA deployment. In their work, the software stacks are installed and configured at deployment time, transferring smaller VMIs through the cloud network. Sun Microsystems [3] proposes an approach to deploy applications in cloud computing environment. Similarly to our approach, their approach uses shell-script files to execute on running cloud nodes at runtime. However, both approaches need experts on virtual image provisioning, who have knowledge about systems and software packages used to compose virtual machine images. While, by using feature models to represent the configuration options, our approach can support both experts and non-experts, who lack knowledge about virtual image provisioning and underlying software systems and dependencies. It can reduce errors and improve the consistency of configurations during the composing of VMIs.

2.6 State of The Art Summary

In this chapter, we have presented background concepts in the domain of cloud computing and how a cloud service is requested or provisioned in the context of providing a virtual machine image as a service.
Then we introduced the fundamental concepts of model-driven engineering that are used throughout the thesis, such as model, modeling, model-transformation, and some relevant tools.
We have presented a background of software product lines, feature modeling, which are methods that are starting to be used to manage the configuration of VMI in cloud computing; and then we described the state of the art of *VMIs configuration management* and VMIs deployment process. Finally, we summarized the related work in Table 2.2.

In summary, the related approaches demonstrate advantages in some cases. However, in the context of VMI provisioning in cloud computing, they still have certain drawbacks:

- Most of the approaches in VMIs configuration management create the appropriate configurations for VMIs and compose the expected VMIs at design time and at the administrator side, before copying and deploying into cloud nodes. It makes it difficult to synchronize the maintenance and modification of the running images as needed when the amount of running cloud nodes is large. In addition, the process of finding the optimal configurations still limited when the user's requirements are diverse and nonspecific.

- Some of the approaches relate to VMIs deployment process support for creating and modifying VMIs at runtime. However, all approaches need experts on virtual image

	Image Specification	Image Deployment	Update and Re-configuration at Runtime	Support Optimization
Dougherty *et al.* [20]	Yes[1]	Create idle VMIs in auto-scaling queue	No	Yes[4]
T. Zhang *et al.* [48]	No	Create typical virtual appliances	No	Yes[5]
Konstantinou *et al.* [29]	Yes[2]	Model-driven approach, abstract deployment plan to defines a composition of configurable VMIs	No	No
Chieu *et al.* [15, 16]; Arnold *et al.* [8]	Yes[3]	Pre-installed VMIs according to the deployment scenarios	No	No
Sethi *et al.* [43]	No	using shell scripts to install, configure software into VMIs at deployment time	Yes[6]	No
Sun Microsystem *et al.* [3]	Yes[7]	using minimum pre-packaged VMIs called "Gold Images"	Yes[8]	No

[1] using feature models
[2] virtual images templates
[3] virtual image templates
[4] using CSPs for energy consumption
[5] by analyzing the probabilistic of virtual appliances frequently used
[6] support a partial update by using shell scripts
[7] using shell scripts
[8] using shell scripts

Table 2.2: Summary of the related approaches in the state of the art

provisioning, who have knowledge about systems and software packages used to compose virtual machine images. Also, the use of static shell-scripts for creating and editing configuration VMIs makes it difficult to re-configure at runtime when the deployment topology of the VMI is complex instead of individual VMI. For example: the deployment topology of VMIs for 3-tiers web applications with VMI is separated for different tiers.

In this thesis, we propose a mechanism for managing VMI configuration in Cloud Computing environments, providing a way to adapt to the needs of auto-scaling and self-configuring virtual machine images, called Model-Driven approach. We use Model-Driven Engineering (MDE) approach for managing VMI configurations and the automated deployment process of VMIs in cloud environment. Following Dougherty *et al.* [20], we consider VMIs as a product line

and use feature models to represent VMI configurations and model-based techniques to handle automatic VMI deployment and reconfiguration. We extend the feature model reasoning engine - SPLAR to handle features with attributes (e.g., installation time, size of software, etc.) and enhance the reasoning process for finding the optimal configurations of VMI according to the user 's selections (both of specific or non-specific selections) from the feature models. Moreover, we will show that applying the model-driven approach helps to reduce the power consumption and adapt to the needs of auto-scaling and self-configuring virtual machine images.

Part II

Contributions

Chapter 3

Feature Modeling for Virtual Machine Image Configuration Management

Contents

3.1 Chapter Overview

This chapter introduces our *feature modeling* approach for managing the configuration of virtual machine images (VMIs). This approach supports the creation of specific VMI configurations according to the user requirements. As discussed in Chapter 1, one of three key issues for the development of VMI provisioning process in cloud computing is defining *"an abstraction level for virtual image configuration management"* . This abstraction should help IT experts of cloud providers to analyse and model the commonalities and the variabilities of VMIs, to specify the product families of VMIs, as well as to create the valid and consistent VMI configurations. Most of the feature model reasoning engines are built to support the reasoning process on the standard feature model. They have the ability to handle the interdependencies and to represent the variability of features. However, in the context of managing the VMI configuration, the features in the feature tree not only contain the *id* and *name* attributes, but also contain additional information, such as: *installation time, size of the package, etc.* Therefore, the existing reasoning engines have not been able to solve the problem that the cloud users require to generate the VMI with optimal configuration according to feature 's attributes (e.g. installation

time, or size of the package, etc.).
This chapter gives the solution for improving the existing feature reasoning engine for searching the optimal VMI configuration with respect to the attributes, for example *installation time* or *size of software*, and then explains why the feature modeling approach is suitable for managing the configuration of VMIs in the provisioning process.
Section 3.2 presents an overview of the virtual machine image configuration with software component install, and the presentation of the VMI configurations as product lines. Section 3.3 describes the feature model reasoning engine that we use for managing VMI configurations. In this section, we give a brief overview of the SPLAR (Software Product Line Automated Reasoning) engine with its limitations (Section 3.3.1) and our improvement SPLAR for the creating the optimal VMI configurations (Section 3.3.3). Finally, Section 3.4 summarizes and discusses the approach.

3.2 Feature Modeling for VMI Configuration Management

3.2.1 An overall architecture of feature modeling

In our approach, VMI configuration product lines are described using feature models. In terms of VMI configuration derivation, a feature model describes:

- The software packages that are needed to compose a Virtual Machine Image, represented as configuration options.
- The rules dictating the requirements, such as dependent packages and the libraries required by each software component.
- The constraining rules, which specify how the choice of a given component restricts the choice of other components, in the same Virtual Machine Image.

The feature modeling approach deals with two models: **The VMI feature model**, **The VMI resolved model**; and a **Product derivation process**.

- **The VMI feature model:** represents the whole product line with all its features as configuration options, their relationships, and constraints which would be used for composing a VMI.
- **The VMI resolved model:** is derived from product derivation process based on user 's selections on the VMI feature model. It includes the selected features and their dependencies.
- **The product derivation process:** generates the VMI configurations from the combination of the VMI feature model and the user 's selections.

3.2.2 The VMI Feature Model

A VMI feature model represents configurations that can be used for composing a VMI. The elements of the VMI feature model are configuration options of a VMI (as features of a feature model), they represent software packages and their dependencies. These elements become

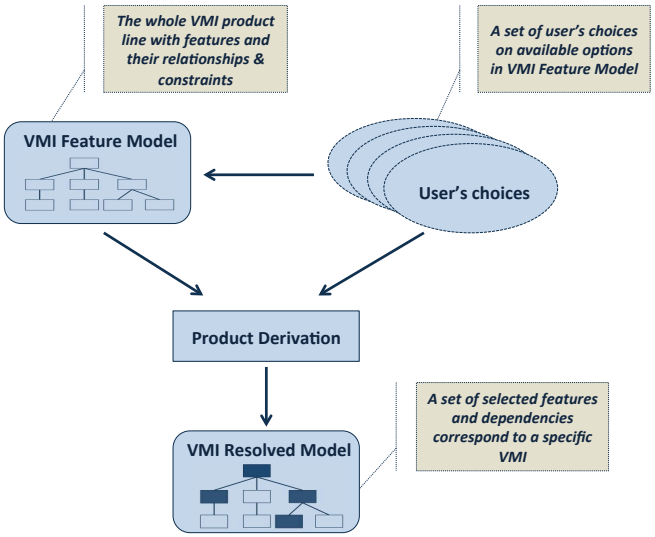


Figure 3.1: An overall architecture of the feature modeling approach

elements of the VMI resolved models, according to the resolutions of the corresponding selection models. The VMI feature model includes the feature tree and the extra constraints. The feature tree represents hierarchical arrangement of configuration options, and the extra constraints are the conjunction arbitrary Boolean formulas for describing the relationship between the feature tree 's elements. The validity of the combination of the extra constraints with the feature tree must be satisfiable. It is validated by a feature model reasoning engine - SPLAR. The detail how we extended the SPLAR engine is explained in later sections.

Figure 3.2 is an example of a VMI feature model. In this model, features and their relationships represent software packages and dependencies. For instance:

- *Operating System* is a mandatory child feature of *Virtual Machine Image*, which must be selected when *Virtual Machine Image* is selected.
- *Operating System* includes two alternative child features: *Windows* and *Linux*.
- When the *Operating System* feature is selected, then either *Windows 7* or *Ubuntu 12.04 LTS* must be selected.
- If the feature *Ubuntu 12.04 LTS* is selected, then features that require *Windows 7* cannot be selected, for instance: *VisualStudio2010*, *JRE 1.6 Windows*, etc.

We can classify the features in a VMI feature model into two types: *category feature* and *package feature*. A *category feature* classifies features, similarly to a folder in file systems. A *package feature* corresponds to a software package which is used for installation, similarly to a file in file systems. It keeps detailed information of a software package, such as: installation time, size of packages, etc. For example in Figure 3.2, *Operating System*, *IDE*, *Database* are category features, while *Eclipse 3.7 Linux*, *VisualStudio2010*, *JRE 1.6 Win* are package features.
VMI feature models are built by IT experts of cloud providers, who have knowledge about

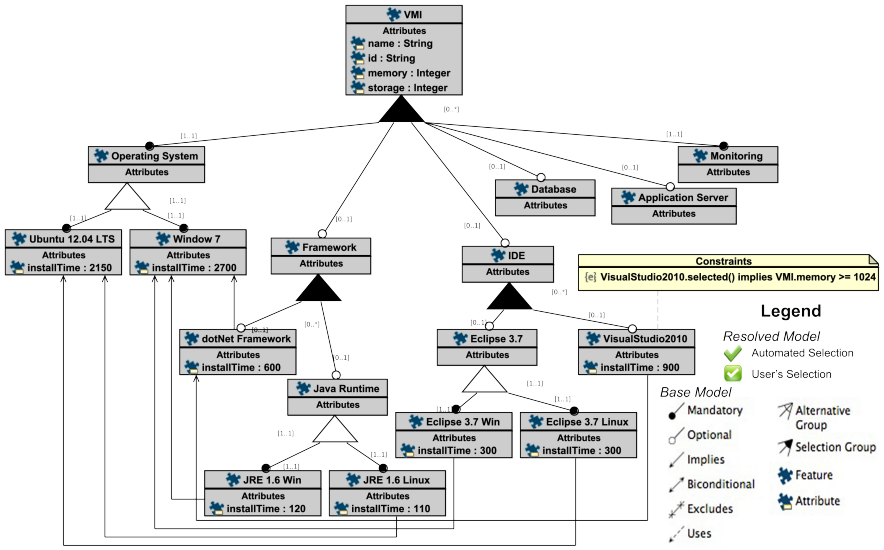


Figure 3.2: A feature diagram representing a VMI feature model of the configuration options

systems and software packages used to compose Virtual Machine Images. The correctness of the VMI feature models relies on the correctness of the feature model that represents them. Many approaches and tools were proposed to automate the analysis of feature models [45, 10, 36]. They offer to validate, check satisfiability, detect "dead" features and analyze feature models. In our implementation, we use the SPLAR engine to validate and to analyse the correctness of the feature model, and to check that the configurations that are derived from a valid feature model are always satisfiable. Details of the mechanism for checking the satisfiability of the feature model are described in Mendonça 's PhD thesis [33].

3.2.3 VMI Product Derivation Process

Product Derivation is the process that is responsible for the creation of the final configuration. It supports the derivation of VMI configurations from the VMI feature model. Figure 3.3 illustrates an interactive scenario of deriving the VMI configuration. In this scenario, cloud customers provide the requirements to the cloud provider, who in turn transforms them into configuration selections. A VMI feature model is provided as input to the derivation process and the VMI resolved model is a result as a complete and valid product specification of a VMI. An automated reasoning process is designed to assist the cloud provider with the reasoning on his selections, and with the propagation of the feature selections throughout the VMI feature model. The selection of each feature is checked and validated by the product derivation process. In each feature selection step, the features connected to the selected feature by a mutually exclusive relationship become unavailable on the VMI feature model for next selections. All of the features that are required by the selected feature are also selected automatically.
Sometimes, a cloud user does not have any specific requirement with respect to a given feature. In these cases, the product derivation process needs a mechanism to generate an optimal solution according to some criteria, such as VMI size, installation time, etc. Generally, the product

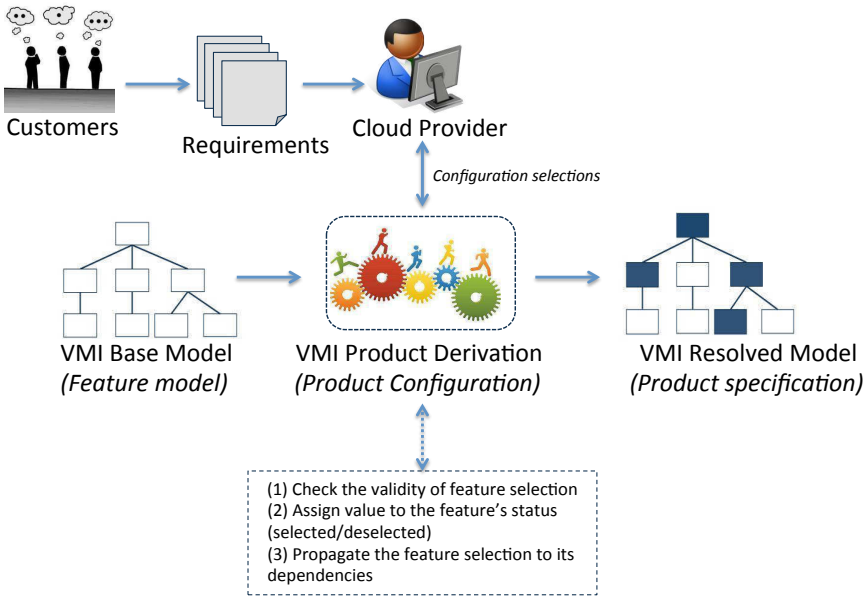


Figure 3.3: VMI Product Derivation Scenario

derivation process is executed in three major steps: (1) Check the validity of the feature selection to ensure that there is no conflict with any previous selections; (2) Assign value to the feature 's value; and (3) Automatically propagate the feature selection throughout the VMI feature model.

3.2.4 VMI Resolved Model

A VMI resolved model stores user's feature choices of the VMI feature model and their dependencies. It is derived from the product derivation process based on user 's selection on the VMI feature model. The product derivation process is responsible to validate user selections and auto-select dependencies based on the feature representation in the VMI feature model. A VMI resolved model corresponds to a specific configuration of a Virtual Machine Image.
Figure 3.4 presents an example of a VMI resolved model that is derived from the VMI feature model. It represents the user 's selections: operating system is *Ubuntu 12.04 LTS*, integrated development environment is *Eclipse 3.7*, and *Apache Tomcat 5.5* for application server. According to the VMI feature model, the dependent features of the user 's selections are also selected by the product derivation process (e.g. *JRE 1.6 Linux*, *IDE*, *Monitoring*, etc.)

3.3 Feature Model Reasoning Engine

3.3.1 Overview of SPLAR

The SPLAR (Software Product Line Automated Reasoning) engine is an open-source library for automated reasoning on feature models. It was developed by Marcilio Mendoca in his PhD work. This library offers SAT and BDD-based components to reason on and to configure feature

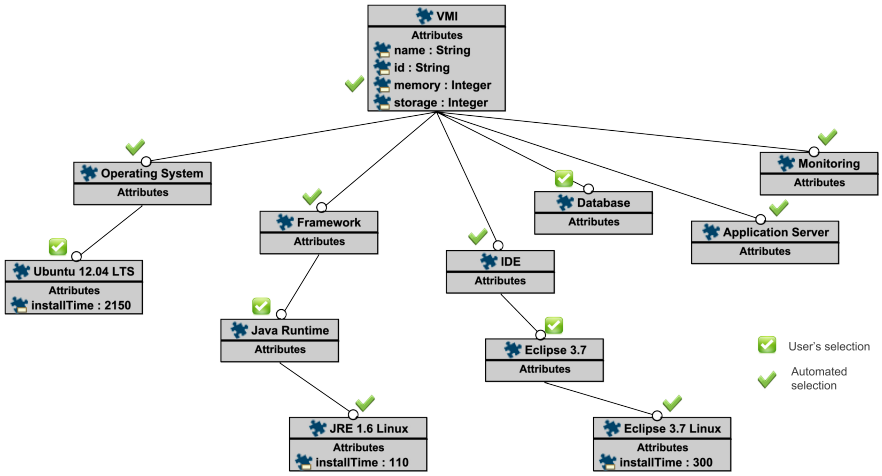


Figure 3.4: A VMI Resolved Model with the User 's Selections and the Automated selections by made the Product Derivation Process

models [37, 34]. It provides a support to validate feature selections, and to generate the valid configurations from selected features and dependencies.
The SPLAR engine proposes a domain-specific constraint solver called **feature tree constraint system** that tailors reasoning algorithms for feature trees [35, 33].

The operations implemented by the **feature tree constraint system** can be summarized as follows:

- **Assigning and resetting the feature values**: These operations help a user to perform the configuration actions on a feature tree such as selecting, deselecting or reseting the status of a feature to available (or uninstantiated).
- **Saving and recovering system states of a feature tree**: These operations are used to save and restore the state of the feature tree. For each instantiated feature in the feature tree, its name and value are saved and linked with a unique identifier. The identifier can be used to restore the feature to a particular saved sate.
- **Propagating the feature value assignment**: This is an important operation to recognize and assign values to the features related to the feature assigned.
- **Checking satisfiability**: This operation checks the satisfiability of a feature tree.

In summary, the SPLAR engine only provide support for reasoning on feature trees; with features containing two basic attributes: *id* and *name*. However, in our approach, a feature on the feature tree represents a configuration option (a software package). It contains information used for optimizing the configuration of a virtual machine image, such as: installation time, un-installation time, package size, cost, etc. Therefore, to use SPLAR as a reasoning engine for the VMI feature models, we extend the original meta-model of feature models supported by the SPLAR engine for representing VMI feature models. The extended meta-model is described in Figure 3.5, Section 3.3.2. Furthermore, we improve SPLAR to allow searching for the optimal configuration of a VMI based upon additional information on the features.

3.3.2 Meta-model for VMI feature model

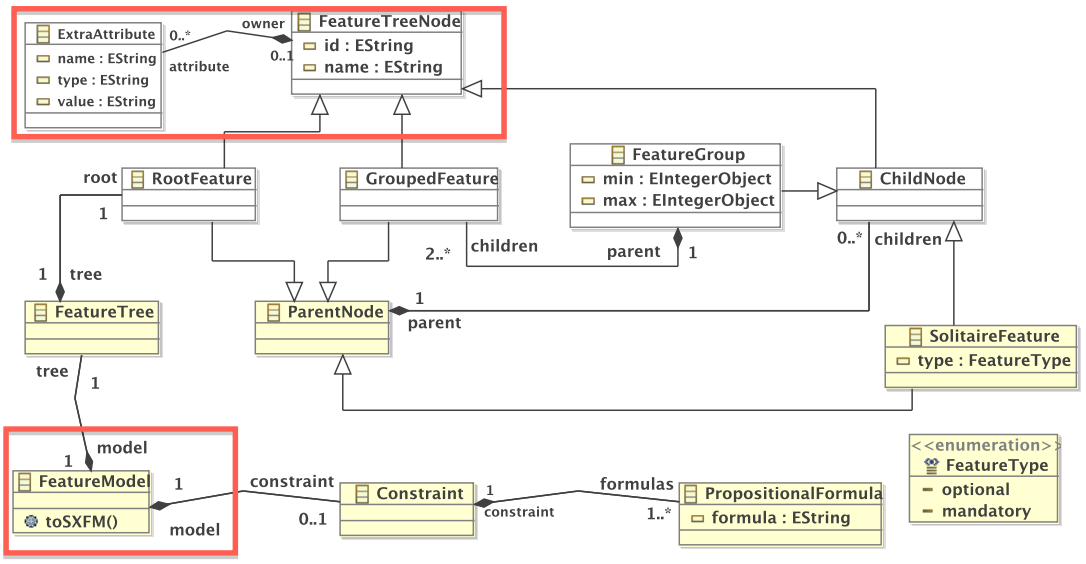


Figure 3.5: An extended meta-model for the VMI feature model

Figure 3.5 shows the extended meta-model for feature models from the original one which is supported by the SPLAR enginer. The red rectangles are extended parts with respect to the original meta-model. In the extended meta-model, the *FeatureModel* element shows that a feature model has a single feature tree (*FeatureTree*) and it may have an extra constraint (*Constraint*). The extra constraint contains one or more Boolean formulas (*PropositionalFormulas*). The *FeatureTree* element represents a feature tree with a single feature root node (*RootFeature*), the root node is a special type of a feature node (*FeatureTreeNode*). It is a parent node of other node, and it does not have a parent. The *FeatureTreeNode* element shows that every node of a feature tree are identified by an *id* and a *name*. It is extended to represent extra attributes (*ExtraAttribute*) of features of the feature tree, and support to users can add any kind of attribute to features, for example: time, cost or energy consumptions, etc. The *SolitaireFeature* indicates the type of a feature is mandatory or optional. The inclusive-or and exclusive-or groups are presented by the *GroupedFeature* element; and the *FeatureGroup* element with attributes *min* and *max* refer to the minimum and maximum cardinality of the group, respectively. The relation between *GroupedFeature* and *FeatureGroup*, *parent* means only feature groups can be parent nodes of grouped features. The relation between *ParentNode* and *ChildNode* elements indicate that the root feature as well as grouped features, mandatory or optional features, and all descendants of *ParentNode*, can be parent nodes of *ChildNode* elements (e.g. feature groups, mandatory or optional features). We also added an operation named *ToSXFM()* to the *FeatureModel* for exporting the VMI feature model to SXFM[1] format and a XML file that is storing the extra attributes of features. SXFM (Simple XML Feature Model) format is used by SPLAR to store the feature model. The advantage of this format is that users can create

[1] http://gdansk.uwaterloo.ca:8088/SPLOT/sxfm.html

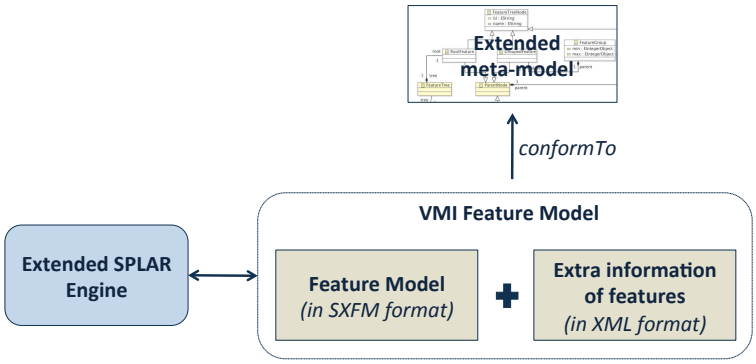


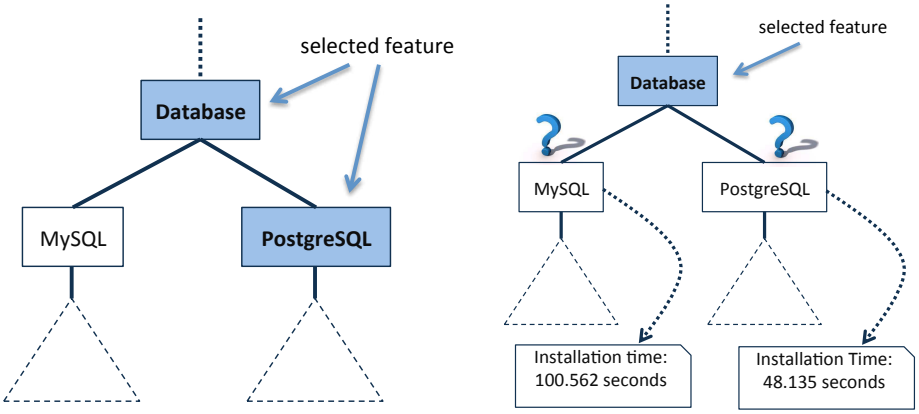
Figure 3.6: SPLART engine woks with VMI feature model

feature model easily and quickly by using a simple text editor.

3.3.3 Optimization in the VMI Product Derivation Process

3.3.3.1 Feature Selection Problem

When cloud users request new VMIs, sometime users do not care much about the details of feature selection. They request VMIs with some generic requirements. For example "- I use Linux Ubuntu, Java programming language, and I need a database" instead of "- I select Linux Ubuntu 9.10, Java JRE 1.6, and I want to use MySQL database", etc. Therefore, cloud providers must have a flexible mechanism in the product derivation process to find an optimal configuration of VMIs that fulfills the user 's requirements and maximizes the benefits. Figure 3.7 presents



(a) A feature selection with a specific requirement

(b) A feature selection with a specific requirement

Figure 3.7: An example feature selection according to the user 's requirement

two examples of feature selection of the feature tree. Consider a cloud user needing a database

server in the VMI. In Figure 3.7a, the user requests exactly *PostgreSQL* database server, so it is easy for the feature selection. However, in Figure 3.7b, the user does not have any specific requirement of which type of database he wants to use. He just requires a database server while the cloud provider has more than one option for a database server (*MySQL*, and *PostgreSQL*), so which one is the best selection for fastest response time? Considering the installation time of these two databases, we can see that the option *PostgreSQL* should be selected because it takes less time for installation than the option *MySQL*.
In another case, suppose that the feature *PostgreSQL* requires another features (e.g. the installation of PostgreSQL needs add-on libraries), so the selection of feature *PostgreSQL* leads to the selection of other libraries. It means the total time for installation of these software would probably be much longer than *MySQL*'s installation time, and the selection of feature *PostgreSQL* would then not be optimal. Therefore, the product derivation process of feature reasoning engine needs a mechanism to make the optimal selection in any non-specific requirement like in the above example.

3.3.3.2 Proposed Solution

For improving the optimization of the product derivation process, we proposed algorithms support the automatic complete the feature selection on the feature tree. Especially, when the user does not have the specific requirement that is described in the above example in Figure 3.7. The process to auto-complete the feature selection consist in two steps: determine the incomplete features in the feature tree and apply the auto-complete process to each of the incomplete selected feature.

⋆ **Checking the incomplete features**

Algorithm 1: An algorithm for checking a feature is incompleted or not

Input: f : is a feature
Output: *true* if f is an incomplete feature

```
1 Function isInCompleted(f:feature):boolean
2     foreach childFeature is child of f do
3         if childFeature is selected then
4             return false
5         end
6     end
7     return true
8 end
```

An *incomplete feature* is a feature in the feature tree which is selected by the user or the propagated actions, with at least one-child feature, but none of the children feature is selected. For example, the feature *Database* in Figure 3.7b is an incomplete feature, and in Figure 3.7a, *Database* is a complete feature. Algorithm 1 describes how to check whether a feature is an incomplete feature or not. The lines 2 to 5 describe that if the feature f has any child feature, which is selected, then it is not an incomplete selected feature, otherwise it is an incomplete selected feature.

⋆ **Completing the VMI configuration selections**

Algorithm 2: An algorithm for completing the selection of VMI configuration

Input: *inFeatureSet* : is a set of incomplete features
Input: *conf* : is a set of selected features (a current configuration)
Output: A set of selected features (a complete configuration)

```
1  Function completeConf(inFeatureSet:feature{};conf:feature{}) :feature{}
2      foreach inFeature in inFeatureSet do
3          fillUnCompleteFeature(inFeature, conf)
4      end
5      return conf
6  end
7
```

Basing on the user 's choices, the Product Derivation engine detects incomplete features and applies the automatic complete selection process to each of them. The procedure described in Algorithm 2 shows that the VMI configuration is filled by applying the automatic complete selection process for every incomplete feature in *inFeatureSet*, and returning the result is a set of selected features for a VMI configuration - *conf*. Detail of how to automatic select a child of the incomplete feature is described in the Algorithm 3.

⋆ **Updating the incomplete feature**

Algorithm 3: An algorithm for updating the incomplete feature

Input: *inFeature* : is an incomplete feature
Output: new selected features, includes a child of f and its requires features

```
1  Procedure fillInCompleteFeature(inFeature:features;conf:feature{})
2      selectedFeature ← 1st child of inFeature
3      foreach childFeature is a child of inFeature do
4          if estimateMinCost(childFeature)<estimateMinCost(selectedFeture) then
5              selectedFeature ← childFeature
6          end
7      end
8      Add selectedFeature to conf
9      Set the status of selectedFeature into selected
10     Add the "selected" propagated features from the selection of selectedFeature to conf
11     if isInCompleted(selectedFeature) then
12         fillInCompleteFeature(selectedFeature, conf)
13     end
14     foreach rf is a feature required by selectedFeature do
15         if isInCompleted(rf) then
16             fillInCompleteFeature(rf, conf)
17         end
18     end
19 end
```

Algorithm 3 implements the process of automatic completion of the selection of incomplete features in the feature tree. Because the incomplete feature could be a parent feature of another incomplete feature, we use a recursive method for the automatic completion of feature selection for incomplete features. The algorithm is stopped when the input feature is a package feature

(or a leaf of the feature tree). The parameters *inFeature* and *conf* are passed as input data of the algorithm. Variable $inFeature$ is an incomplete feature which needs to be considered, and $conf$ is a set of the selected features (by the user 's choices or propagation actions) at the current step. The variable $selectedFeature$ represents a child of the feature $inFeature$ which should be selected. Firstly, $selectedFeature$ is assigned by the first child of $inFeature$ (line 2). Secondly, we consider all child features of $inFeature$, if a feature is found, which is satisfies the optimization constraint, then it is assigned to the $selectedFeature$ (lines 3-5). The optimization constraint of the estimation of minimum costs is used to guarantee that the $selectedFeature$ is the best choice at the moment. The detail of the estimation of minimum cost ($estimateMinCost$) is described in the Algorithm 4. When the feature $selectedFeature$ is determined, it is selected and added to the current configuration - $conf$. The features that are required by the selection of $selectionFeature$ also added to $conf$ (lines 8-10). The recursive calls are made to examine each of the new selected features ($selectedFeature$, and its required features) if they are incomplete features(lines 11-18).

⋆ **Estimating the minimum cost of a feature**

Algorithm 4: An algorithm for estimating the minimum cost if a feature f is selected

```
   Input: f : is feature which is need to estimate the minimum cost
   Input: conf: a set of selected features (a current configuration)
   Output: the minimum cost if the feature f is selected
 1 Function estimateMinCost(f:feature, conf:feature{}):double
 2     cost ← f.cost
 3     foreach r is a required feature by f do
 4         if r is not in conf then   /* selected by user or a propagated action */
 5             cost ← cost + estimateMinCost(r)
 6         end
 7     end
 8     if f has child features then
 9         if f has mandatory child features then
10             foreach childFeature is a mandatory child feature of f do
11                 cost ← cost + estimateMinCost(childFeature)
12             end
13         else
14             minFeature ← 1st child of f
15             foreach childFeature is a child feature of f do
16                 if estimateMinCost(childFeature) < estimateMinCost(minFeature)
                   then
17                     minFeature ← childFeature
18                 end
19             end
20             cost ← cost + estimateMinCost(minFeature)
21         end
22     end
23     return cost
24 end
```

Estimating the installation time, package size or operational cost of each feature selection is very important to create an optimal VMI configuration. It helps to select a child of an incomplete feature which has the minimum time, size or operational cost. We provide a recursive algorithm (see Algorithm 4) named *estimateMinCost* for estimating the minimum cost of a feature if it is selected. Depending on the optimization requirements of a VMI configuration, the cost could be the installation time, package size or operational cost. For example, if the cloud provider wants to focus on the response time, then the Product Derivation process has to find the best solution based on the installation time of software packages. Therefore, the optimal function *estimateMinCost* works with the installation time attribute of the feature. Line 2 of the Algorithm 4 can be re-written as: $cost \leftarrow f.installTime$.
The Algorithm 4 helps the Product Derivation process to automatic select the best solution in terms of minimum cost. It examines the total cost of the candidate feature and its dependencies (lines 3-7). In addition, if the feature has more than one child features, which are the candidates for the selection, then the algorithm is called recursively to examine the child feature that has a minimum cost in two cases. Firstly, if the feature f has any mandatory child features, then they are automatically selected, and the total estimation minimum cost of f - *cost* will be added the estimation minimum cost of its mandatory child features (lines 9-12). Secondly, all child features of f are examined to find a feature with the smallest cost. The local variable $minFeature$ is temporary assigned by the first child feature of f (line 14). The loop through all child features of f allows to compare each value of $estimateMinCost$ function to the $estimateMinCost$ value of $minFeature$. If there is any feature with the value of $estimateMinCost$ function that is smaller than the $minFeature$'s $estimateMinCost$ then that feature is assigned to the $minFeature$. Thus, the feature $minFeature$ which is obtained at the end of the loop is the best choice (lines 15-19), and the estimation minimum cost of f - *cost* accumulates the estimation minimum cost of $minFeature$ (line 20). At the end of algorithm, the value of *cost* is returned as the value of $estimateMinCost$ of the feature f. This is an estimation minimum cost of the feature f if it is selected.

⋆ **The correctness of the algorithms**
Algorithms 3 and 4 are recursive procedures. They are implemented on the feature tree with a finite number of feature nodes. To verify the correctness of the algorithms, we need to prove that with any given feature in the valid feature tree, the algorithms will be terminated after a finite number of recursive calls. This proof will use a common technique for proofs in recursive programs called an inductive proof. Because the Algorithm 4 is called in the Algorithm 3, therefore, for easy tracking, we present the proofs of the correctness of the algorithms in order Algorithm 4 and then Algorithm 3.

Proof 1: *Verifying that for any given feature in a feature, and corresponding current configuration, the Algorithm 4 will terminate and return a minimum estimation cost.*

- **Assumptions**
  In the proofs, we refer to the four recursion points of the program as R1 (line 5), R2 (line 11), R3 (line 16), and R4 (line 20) respectively. The program will carry the implicit assumption that the feature tree is valid and with a finite number of features. It does not contain any inclusive cycle relationship (e.g. A require B, C; C requires D, and D requires A) and conflict inclusive or exclusive relationship (e.g. A requires B; B requires C, and C excludes A) between the features.

- **Base case proof**
  The algorithm is terminated when f is a leaf and an independent feature. The estimation of minimum cost is the cost of feature f ($estimateMinCost = f.cost$)
- **Inductive step proof**
  - Each iteration of the program, the recursive is called by either R1, R2, R3, or R4.
  - R1 will only occur when the feature f has some require features. The number of features that are required by f is finite, and the valid feature tree does not contain any inclusive cycle relationship between the features(e.g A require B, C; C requires D, and D requires A). Thus, the number of the iteration calls R1 is finite, and the cost increments by the sum of estimate cost of the require features of f.
  - R2 or R3 will only occur when the feature f has children. R2 occurs when f has mandatory features, and the number of mandatory feature is finite. In this case, the recursive called for these mandatory features, and the estimate cost of f increments by the sum of estimate cost of the mandatory child features of f. Otherwise; R3 will occur when f has children, and the number of child feature is finite, and it is not too big, but no one of them is a mandatory feature. The first child of f is considered as a pivot-feature, the loop makes the comparisons between the estimate cost of this pivot-feature to other child features of f. Therefore, the pivot-feature at the and of the loop is a child feature of f that is led to the minimum cost if it is selected (R4). Because of hierarchy structure of the tree then the recursive calls will reach to the termination when the input feature is a leaf feature.

We have now proven that with the given assumptions, the Algorithm 4 will terminate and return the estimation of minimum cost of a feature if it is selected.

Proof 2: *Verifying that for any given incomplete feature in a feature tree, the Algorithm 3 will terminate.*

- **Assumptions**
  We refer to the two recursion points of the program as R1 (line 12) and R2 (line 16) respectively. Like the assumption of the Proof 1, the program will carry the implicit assumption that the feature tree is valid with a finite number of features. It does not contain any inclusive cycle relationship (e.g. A require B, C; C requires D, and D requires A) and conflict inclusive or exclusive relationship (e.g. A requires B; B requires C, and C excludes A) between the features.
- **Base case proof**
  The algorithm is terminated when the selected child feature ($selectedFeature$) of $inFeature$ is a leaf and an independent feature, and the new selected features are added to $conf$.
- **Inductive step proof**
  - Each iteration of the program, the recursive is called by either R1 or R2.
  - R1 will only occur when the selected child feature of $inFeature$ ($selectedFeature$) is an incomplete feature. Because the number of features of the feature tree is finite, then the length of the path from the current feature ($inFeature$) to the leaf feature is finite. Therefore, the number of the iteration calls R1 is finite.

– R2 will only occur when the selected child feature of $inFeature$ ($selectedFeature$) has required features. In this case, the recursive called for these required features of $selectedFeature$. Because the number of features that are required by $selectedFeature$ is finite, and the valid feature tree does not contain any inclusive cycle relationship between the features(e.g A require B, C; C requires D, and D requires A). Thus, the number of the iteration calls R2 is finite.

From the above explanation, we have now proven that with the given assumptions, the Algorithm 3 will terminate with any incomplete feature. Figure 3.8 is an example of the Product

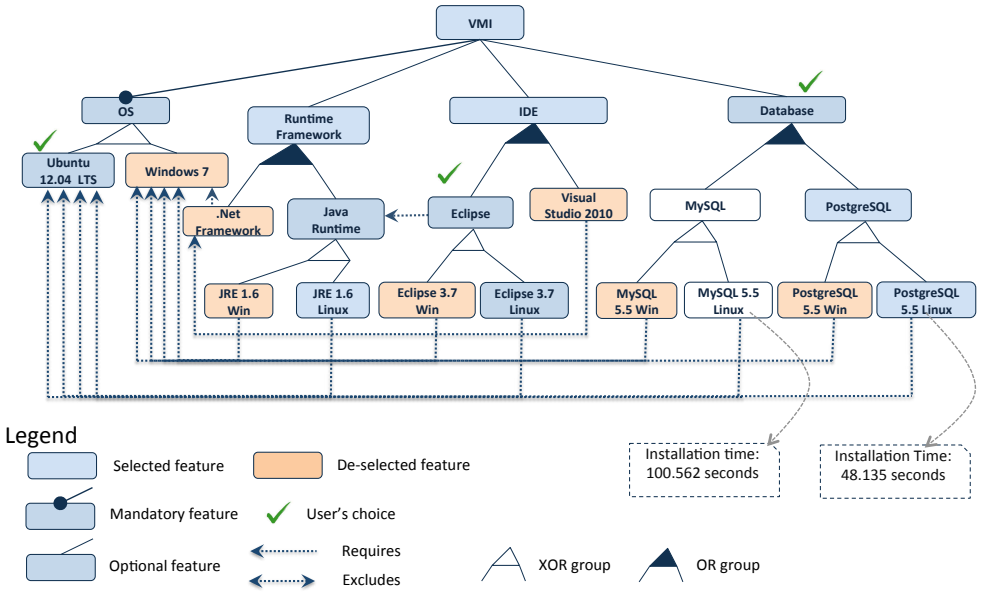


Figure 3.8: An example of the VMI feature model with the selected features

Derivation process applied for finding the optimal VMI configuration according to the installation time of software packages. In this example, user selects three features *Ubuntu 12.04 LTS, Eclipse,* and *Database*. According to the selection rules of the feature model, the features *Ubuntu 12.04 LTS* and *Windows 7* are mutually exclusive, so that when the feature *Ubuntu 12.04 LTS* is selected then *Windows 7* and its dependent features (e.g. *.Net Framework, Eclipse 3.7 Win, etc.*) are de-selected. The development environment Eclipse needs Java's runtime framework, so when the feature *Eclipse* is selected then *Java Runtime* is also selected. Because the features *Java Runtim, Eclipse,* and *Database* are *incomplete features*, then the Algorithm 3 is applied for these features. Especially, in the case of *Database* feature, it has two child features *MySQL* and *PostgreSQL*. However, by applying the Algorithm 4, the feature *PostgreSQL* is selected because its estimating minimum cost is smaller than *MySQL*'s estimating the minimum costs.

3.4 Chapter summary

In this chapter, we have presented our approach of using feature modeling for managing the VMI configurations. By using feature modeling methodology, the VMI configurations are con-

sidered as the VMI Product Lines. It helps cloud providers in analyzing, and modeling the commonalities and variabilities of VMIs. The feature modeling methodology also supports the construction of *an abstraction level for virtual machine image configuration management*.
We have also introduced the use of a feature reasoning engine - SPLAR and its limitations in the derivation of VMI product. Hence, we have presented the way that we extended SPLAR engine to adapt to the requirements of the product derivation process in terms of the VMI configuration management. We have described the extension of the feature tree 's metamodel to handle the VMI Feature Model, and adding the algorithms for finding the optional configuration of a virtual image. In summary, this chapter details the use of feature modeling in the VMI configuration management and give solutions for solving the challenges which were listed in Chapter 1:

- *Handling the interdependencies of software packages*
- *Modeling the commonalities and variabilities of VMI 's configuration options*
- *Finding the optimal configuration of a VMI and guaranteeing the validity and consistency of the derived VMI configurations*

Chapter 4

Model-driven engineering for VMIs deployment and reconfiguration at runtime

Contents

4.1 Overview of chapter

In this chapter, we present the approach of using model-driven engineering for the VMIs deployment in the provisioning process and the reconfiguration of the images at the runtime. The approach focuses on the process of creating the images and installing the needed software package from an initial template image rather than copying a virtual disk image as does the traditional approach. It uses models to encapsulate the series of procedural operations of the deployment and reconfiguration of VMIs provisioning process.
The chapter is organized as follows. Section 4.2 introduces an overview of the model-based VMIs provisioning process. Section 4.3 presents about the deployment of VMIs. Section 4.4 describes how model@runtime is used for the reconfiguration of VMIs at runtime and additionally, how the feature modeling approach (described in Chapter 3) is used in the reconfiguration of VMIs. Finally, Section 4.5 gives a discussion and summarizes the chapter.

4.2 The model-driven VMIs provisioning process

Moded-driven approach is a software technique that focuses on creating and using domain models rather than on the computing concepts. A domain model provides an abstraction representation of the knowledge and activities that manage a particular application domain. In the context of VMIs provisioning process in cloud computing, the model-driven approach provides a systematic use of models as primary artifacts throughout the deployment, installation and reconfiguration of images and software packages.

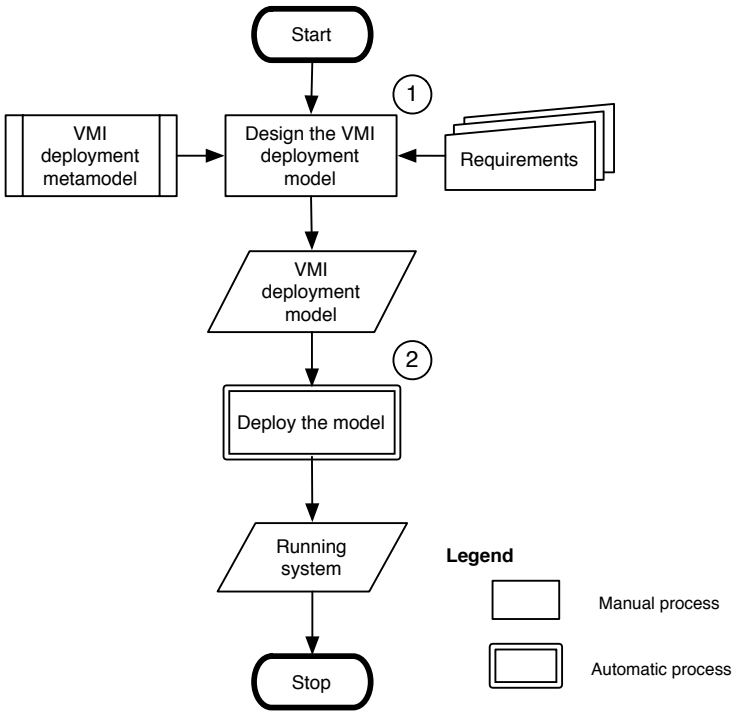


Figure 4.1: A model-based VMIs deployment process

While the traditional approach binds software components together at the time that the template image is created, the model-driven approach binds components together at boot time. Figure 4.1 shows the life-cycle of the model-based VMIs provisioning process. It includes two processes: (1) *Design the VMIs deployment model* is a manual process that is handled by the users and (2) *Deploy the VMIs deployment model* is an automatic process that is performed by the pre-define procedural operations.
In the process 1, after analyzing the user 's requirements, the cloud providers create the appropriate configuration of the images and design the deployment model of VMIs with respect to the VMIs deployment metamodel. The outcome of this process is a VMIs deployment model. It contains the presentation of the software to be installed in a VMI, and also shows the connections of the software installed in the different VMIs. In the process 2, the created VMIs

deployment model is deployed to create a system with the desired virtual machines. In this process, the initial template images with the well-suited operating systems (according to the configurations that were defined in the previous step) are booted to the cloud nodes. When the booting is complete, the VMIs deployment model is deployed and executed direct on the running VMIs. The installation and configuration of software components occur inside the running VMIs.

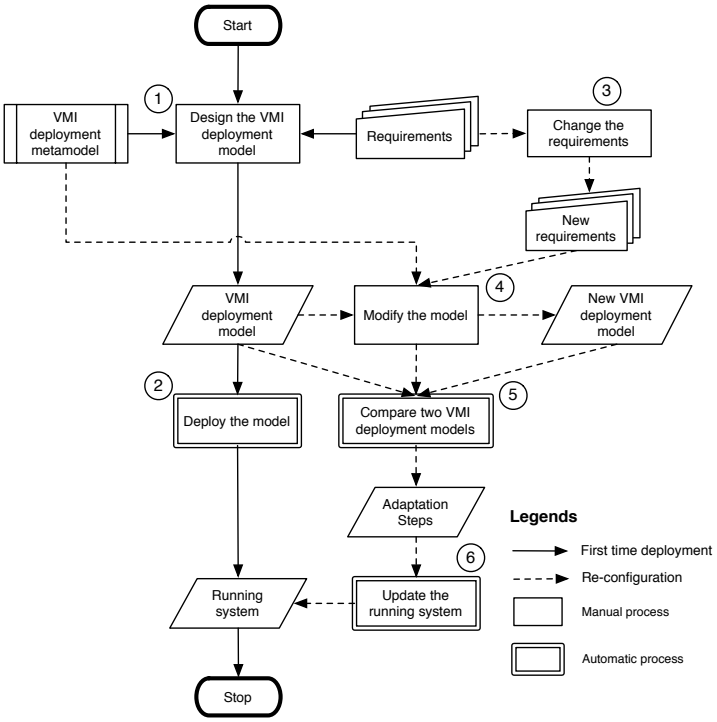


Figure 4.2: The model-based VMIs deployment and reconfiguration at runtime

At runtime, the modification of these VMIs is implemented by changing the configuration of VMIs in the VMIs deployment model and re-deploying the new deployment model to the running VMIs. Figure 4.2 shows the model-based VMIs deployment and reconfiguration at runtime process. The solid lines represent the process flows of the first time deployment of the model (equivalent to the representation in Figure 4.1), and the dashed lines represent the process flows of the reconfiguration of VMIs at runtime. At runtime, the change of user 's requirements (process 3) leads to the creation of a new VMIs deployment model (process 4) the new VMIs deployment model is derived from the existing one according to the change of requirements and it also conforms to the VMI deployment metamodel. The new VMIs deployment model is compared to the current model to determine the differences between them and propose the adaptation steps (process 5). Finally, these adaptation steps are applied to the running system (process 6) to change it into a new system with the desired virtual machines. In the model-driven approach, the cloud providers must define and record the steps needed

to create the VMI in such a way that they can reuse to perform an unattended install and configuration of the desired virtual images for every instance that is created in the future, and we call it the model. Once a model is created, the model itself can be manipulated to create different types of VMIs. In the context of VMIs provisioning in cloud computing, a model is a specification of the VMIs deployment. It specifies a VMIs deployment topology, VMIs that include the software components, and the connections between the software of different VMIs. It represents the desired VMI configurations and how the VMIs work together at runtime. The detail of the VMIs deployment and reconfiguration is explained in next sections.

4.3 The VMIs deployment

By using model-driven engineering approach, we aim at the creation of the deployment models of VMIs in cloud computing as the platform-independent model (PIM) that are independent of the specific technological of the cloud platform used to implement them. With the use of a VMIs deployment metamodel, our approach supports to create the flexible and valid VMIs deployment models that are independent from the cloud platforms. In the later sections, we present two metamodels for the VMIs deployment in: (i) a single cloud system, and (ii) a federated cloud system; and the key elements of the metamodels with their abstract definitions and implementations. For more clarity, we explain the terms of *single cloud* and *a federated cloud* in our approach as following:

- **A single cloud**: is a cloud system provided by a service provider with a specific technological platform. It could be a private cloud, a public cloud or a hybrid cloud. Users can access, interact and monitor the virtual machine in the cloud by using specific APIs that are provided by the provider. Examples of a single cloud system are: *Amazon EC2*, *Grid5000*, *IBM SmartCloud*, *OpenNebula*, etc.

- **A federated cloud**: is also called *multiple clouds system* or *inter-cloud system*. It is a system that contains various cloud systems with different technological platforms. For example, a federated cloud system could be the union of the common parts (e.g VMIs provisoning at IaaS level) of three different cloud platforms: *Amazon EC2*, *OpenNebula*, *IBM SmartCloud*. In the other words, in our approach, a federated cloud is the deployment and management of multiple cloud computing services to fit business needs.

4.3.1 VMIs deployment metamodels

4.3.1.1 A metamodel for the VMIs deployment in a single cloud system

Figure 4.3 presents an example of the VMI deployment metamodel for the VMIs deployment in single cloud systems. This metamodel is a precise definition of the construction and rules for the creation of semantic models of the VMIs deployment. It is designed with the Eclipse Modeling Framework (EMF)[1]. The *VMIDeployModel* entiy is a root element of the metamodel. It is comprising all the other entities. Each *VMIDeployModel* instance specifies a VMI deployment model. A VMIs deployment model contains the number of virtual machines and connections

[1] http://www.eclipse.org/modeling/

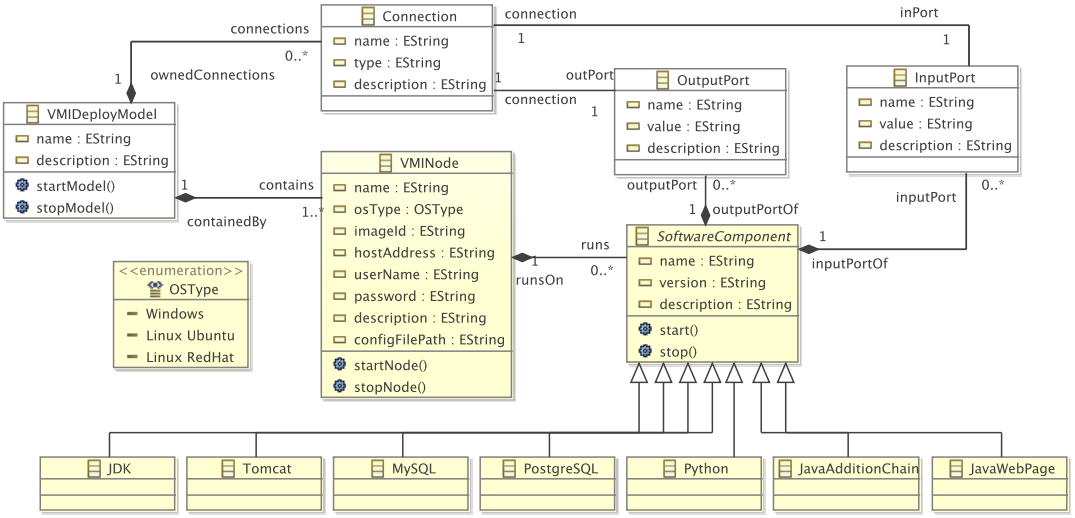


Figure 4.3: The VMIs deployment metamodel for a single cloud system

between their installed software. The *VMIDeployModel* element shows that a VMI deployment model contains at least one virtual machine (*VMINode*) and it may have some connections (*Connection*) that define the links between them (through software installed in machines).
The *VMINode* element is an abstract representation of a concrete virtual machine that runs in the cloud system. It supports users to define the expected configuration of the virtual machine (e.g. *username*, *password*, *imageId*, etc.). The *VMINode* can be encountered in different types of operating systems, as enumerated in *OSType*, indicating the *VMINode* is a cloud of *Amazon EC2*, *Grid5000* or *IBM* cloud; and runs a specific operating system (*Windows*, or *Linux Debian* or *Linux RedHat*). It also provides two operations: *startNode* for starting the virtual machine, and *stopNode* for terminating the virtual machine. The virtual machine (*VMINode*) can contain several software packages. A software package is denoted by the *SoftwareComponents* element. The *SoftwareComponent* element contains three operations: *start* for installing the software package, *updatePort* for updating the needed configuration data that it provides to or requires from the others, and *stop* for removing the software packages.
The *InputPort* and *OutputPort* entities are the abstract representations of the components that encapsulate the needed configuration information of software. The information is stored by the *InputPort.value* or *OutputPort.value* attribute. A port is a part of a software component model, and a software component model may have many ports. An input port (*InputPort*) contains the information required by its owner (a software package) from another software while an output port (*OutputPort*) contains the information that its owner (a software package) provides to the other software.
The *Connection* element is the abstract representation of a component that defines the connection between two software. It associated with two ports (a *InputPort* port and a *OutputPort* port). It assigns the value of its *OutputPort* port to the corresponding *InputPort* port. A VMIs deployment model may have many connections.

4.3.1.2 A metamodel for the VMIs deployment in a federated-cloud system

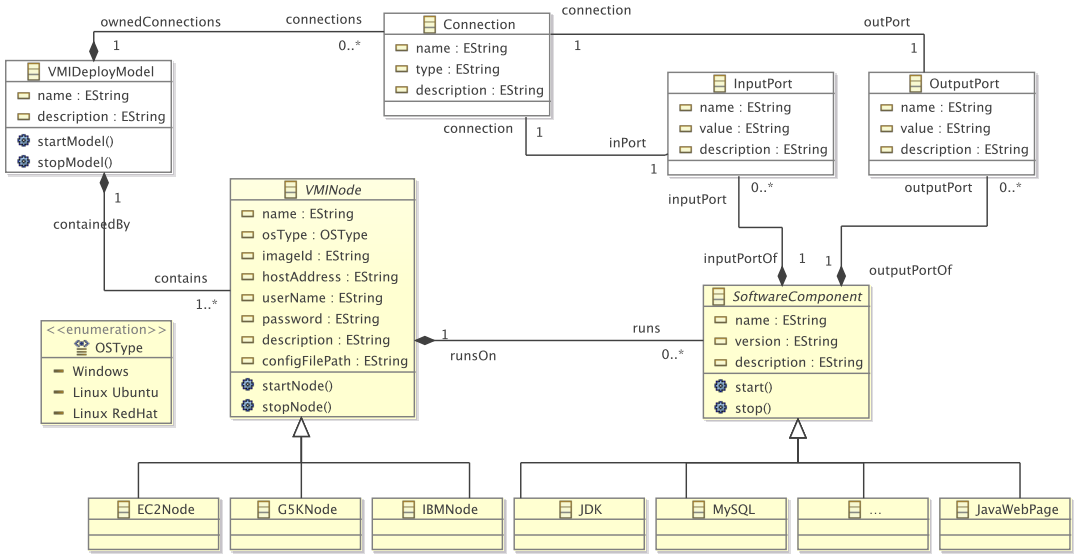


Figure 4.4: The VMIs deployment metamodel for a federated cloud system

The major issue of the VMIs deployment in a federated cloud system is the ability to work with various cloud systems that have different technological platforms. Therefore, it is needed to build a provisioning framework at high level abstraction to be able to interact with these various platforms.

Thanks to the advantage of model-driven engineering, we create a metamodel for the VMIs deployment in a federated cloud by extending the VMIs deployment metamodel for a single cloud. Because the abstract representation of the specific technological issues of a cloud is defined by the *VMINode* element, so that the new metamodel inherits almost all elements from the metamodel in the case of a single cloud VMIs deployment. We focus on extending the *VMINode* element to be able to define the abstract representation of different clouds in the federated cloud system.

Figure 4.4 shows an example of a metamodel for a federated cloud. In this federated cloud, the system includes three different cloud platform: Amazon EC2, Grid5000 and IBM SmartCloud. By extending the *VMINode* element, we distinguish three different children of it: *EC2Node* for Amazon EC2, *G5KNode* for Grid5000, and *IBMNode* for IBM cloud. These element are pre-defined in the protocols for interacting to the corresponding cloud platforms to manage the virtual machines, such as: lauch the VMIs, access to the VMs or terminated the VMs.

In general, the VMIs deployment in a single cloud can be seen as a specific case of the VMIs deployment in a federated cloud when there is only one cloud platform or the clouds use the same technological, APIs for interacting and managing the virtual machines. For more detail, in the later sections, we presents how the VMIs deployment model, VMI node and software component are represented at the abstraction level by the elements: *VMIDeployModel*, *VMINode* and *SoftwareComponent*.

4.3.1.3 VMIDeployModel

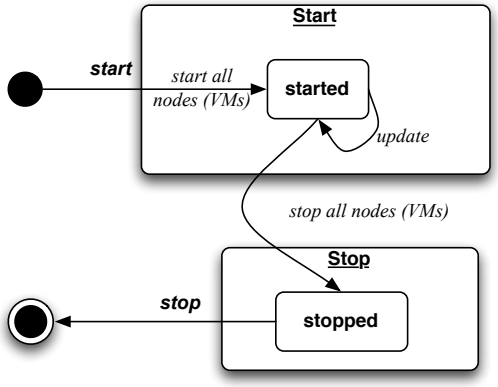


Figure 4.5: Life cycle of a VMIDeployModel instance

The *VMIDeployModel* entity of the VMIs deployment metamodel represents an abstraction level of a VMIs deployment model. It is a pre-definition of the configuration of VMIs and the deployment topology of these images. The instance of this metamodel is a concrete deployment model of VMIs in the specific cloud platforms.
Figure 4.5 represents the life-cycle of the VMIs deployment model execution. It models the behaviour of a VMIs deployment model, specifying the events that a model goes through during its lifetime. The VMIs deployment model can be in one of two states: *started* or *stopped*. It can respond to the events: *"start all nodes"*, *"update"* and *"stop all nodes"*. Not all events are valid in all states; for example, if the model is not *started* or the model is *stopped* then we cannot *update* until we start it. If the model is in *started* state, the applying of the *"stop all the nodes"* event will transit the state of model into *stopped*. Notice that the *VMIDeployModel* instance is started or stopped successful when all virtual machines are started or stopped successful accordingly.

Figure 4.6 represents the abstract definition of the *VMIDeployModel* entity. It defines the attributes and the behaviors (*start* and *stop* operations) of the VMI deployment models at the abstraction level. The implementation of the *VMIDeployModel* is represented in Listings 4.1, 4.2.
As described above, when the VMIs deployment model is started then all the virtual machines (represented by *VMINode* entities) are started. Because the executions of the virtual machines are independent together, they can be done simultaneously. We use multiple thread technique in Java to perform these executions. Listing 4.1 is the sample Java code to create a thread to handle the *start* or *stop* actions of a virtual machine image according to the input action request. The implementation of the *start* and *stop* events are defined by the *startModel* and *stopModel* procedures in the Listings 4.2. In these procedures, we look for all *VMINode* instances in the model and call their *start* or *stop* events and wait until the execution of all these instances are finished.

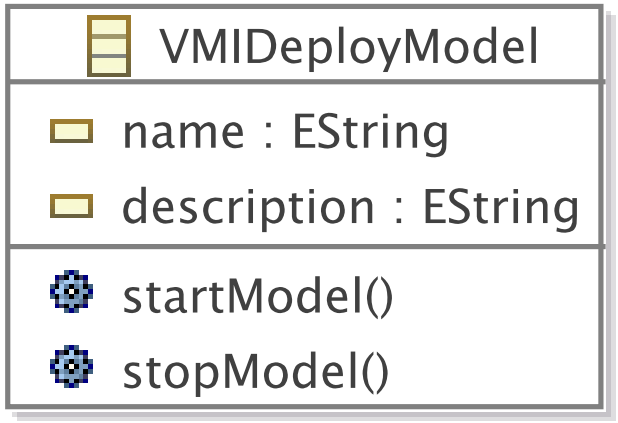


Figure 4.6: The abstract definition of the VMI deployment model

Listing 4.1: Executing multiple VMINode in parallel

```
public static class MyRunnable implements Runnable {
    private final VMINode node;
    private final String action;
    MyRunnable(VMINode node, String action) {
        this.node = node;
        this.action = action;
    }
    @Override
    public void run() {

        try {
         if (action.equals("start")){
            System.out.println("\nStart node: "+node.getName());
            node.startNode();
         }else
            if (action.equals("stop")){
               System.out.println("\nStop node: "+node.getName());
               node.stopNode();
            }else{
               System.out.println("\nInvalid action command (start / stop)!");
            }
        } catch (Exception e) {
        }
    }
}
```

Listing 4.2: The *startModel* and *stopModel* procedures of a VMI deployment model

```
@Override
```

```
public void startModel() {
   EList <VMINode> vmiNodes = this.getContains();
   ExecutorService executor = Executors.newFixedThreadPool(vmiNodes.size());
   for (int i =0; i< vmiNodes.size();i++ ){
      VMINode node = vmiNodes.get(i);
      Runnable runner = new MyRunnable(node, "start");
      executor.execute(runner);
   }
   executor.shutdown();
    while (!executor.isTerminated()) {
      }
}
@Override
public void stopModel() {
   EList <VMINode> vmiNodes = this.getContains();
   ExecutorService executor = Executors.newFixedThreadPool(vmiNodes.size());
   for (int i =0; i< vmiNodes.size();i++ ){
      VMINode node = vmiNodes.get(i);
      Runnable runner = new MyRunnable(node, "stop");
      executor.execute(runner);
   }
   executor.shutdown();
    while (!executor.isTerminated()) {
      }
}
```

4.3.1.4 VMINode

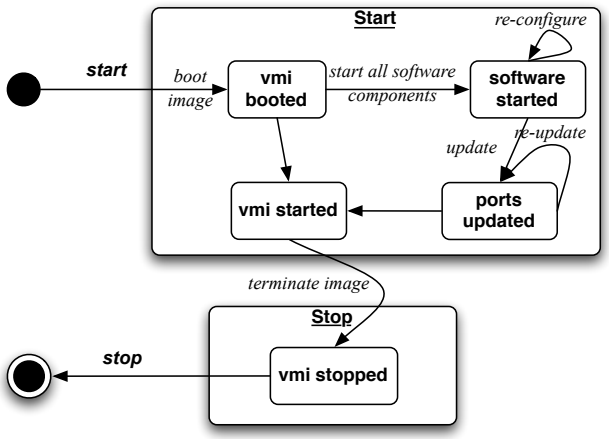


Figure 4.7: Life cycle of a VMINode instance

The *VMINode* is an abstract representation of the virtual machines in the cloud system.

It carries out the tasks: reserving the resources and booting the images, installing software packages into the images and terminating images to release the resources. Figure 4.7 shows the life-cycle of a VMI node (as a *VMINode* instance). It models the behaviour of a virtual machine, specifying the sequence of events that a VM goes through its lifetime.
According to the Figure 4.7, a virtual machine can be in one of the following states: *"vmi booted"*, *vmi started*, *"software started"*, *"ports updated"* and *"vmi stopped"*. It can respond to the events: *"boot image"*, *"start all software components"*, *"re-configure"*, *"update ports"* and *"terminate image"*. Not all events are valid in all states, they have to follow the order that defined in the figure, for example: the events *"start all software components"*, *"re-configure"* and *"update ports"* cannot be applied before the event *"boot image"* and after the event *"terminate image"*.
The states and events of the *VMINode* 's life-cycle are occurred in two operations: *start* and *stop*.

- **start:** If the *VMINode* instance is requested to start then the *"boot image"* event is executed. The resources (e.g. memory, storage, processor, etc.) are reserved according to the input parameters and a selected VMI is booted. When the VMI boot finishes, the virtual machine state is *vmi booted*. The execution of the model will continue in one of two possibilities:
  - If there are some *SoftwareComponent* entities assigned to the *VMINode* in the VMIs deployment model then the event *"start all software components"* is executed and the execution continues to install the software packages that are represented by *SoftwareComponent* entities. When the installation of the software packages is finished the state of the VMI node is transited to *"software started"* and the event *"update ports"* is applied automatically, and then the state of the virtual machine moves to *"port updated"*. Within the execution of the event *"update ports"*, the ports (*InputPort* and *OutputPort*) of software packages are looked up and processed for the connections between them. When all these processes are finished, the virtual machine is successfully started with the state of the VMI node is *"vmi started"* and the virtual machine is ready to use.
  - If there is no *SoftwareComponent* entities assigned to the *VMINode* in the VMIs deployment model, then the state of the VMI node is transited into *"vmi started"* automatically, and it is ready to use. In this case, the virtual machine starts with a clean VMI. It contains an operating system and default software packages.
- **stop:** If the *VMINode* instance is requested to stop then the event *"terminate vmi"* is applied to the running virtual machine. It will shutdown the virtual machine and delete the image. When the termination is finished then the virtual machine is stopped and the state of the VMI node is transited to *"vmi stopped"* and all resources are released.

In Figure 4.8, we define the abstract definition of the *VMINode* entity with the properties used to interact with the specific cloud platforms and two operations (*startNode* and *stopNode*) for starting and stopping the virtual machine. A concrete VMINode instance specifies a virtual machine running in a specific cloud platform. Therefore, it requires the specific APIs to interact with the cloud platform to handle the VMs operations. For the prototype implementation of this thesis, we classify the VMINode instances into different types according to the supported

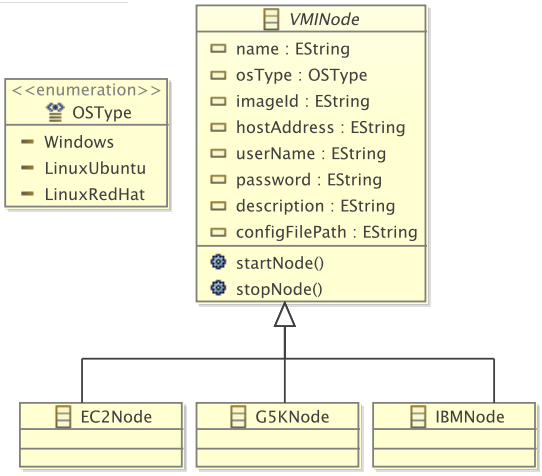


Figure 4.8: An example of the *VMINode* for a virtual machine in specific cloud platform

cloud platforms. Figure 4.8 shows an example of the VMINode instances classification. The *EC2Node*, *G5KNode*, and *IBMNode* classes are children of *VMINode* class. The instances of these classes are abstract representations of the VMs in specific cloud platforms: Amazon EC2, Grid5000 and IBM Smart Cloud accordingly.
Listing 4.3 is an example of the *start* operation implementation of a VMI node for starting a virtual machine in Amazon EC2 cloud platform. We have to define the additional tools (named *AmazonEc2Utils* in Line 2) that use the specific APIs to handle the operations in the Amazon EC2 cloud platform. The operations *start* and *stop* are represented in the procedures *startNode* and *stopNode*. In the *startNode* procedure, we define how to create and boot a virtual machine (also called an *EC2 instance*) in Amazon EC2 (Line 9 to Line 18). When the machine is successfully booted (Line 20), we can update or re-configure the virtual machine to ensure that the machine is working well and is ready to use, such as software installation, etc (Line 21 to Line 29). After that, if there are some *SoftwareComponent* entities presented for this machine, the equivalent software packages will be installed by the calling *start* method (Line 33 to Line 36). When the installations of the needed software packages are complete, a method named *updatePorts()* is called to update the *InputPort*, *OutputPort* and the *connections* between these software (Line 37).

Listing 4.3: The *startNode* procedure for starting a VMI in the case of Amazon EC2 cloud platform

```
1 public class EC2NodeImpl extends VMINodeImpl implements EC2Node {
2   private AmazonEC2Utils appUtils;
3   private Instance instanceEC2;
4   protected EC2NodeImpl() {
5     super();
6   }
7   @Override
8   public void startNode(){
9     Properties prop = new PropertiesFile().loadProFile(this.getConfigFilePath());
```

```
10        String endpointRUL = prop.getProperty("endpointURL");
11        String credentialsFile = prop.getProperty("credentialsFilePath");
12        appUtils = new AmazonEC2Utils(credentialsFile, endpointRUL,this.getName());
13        try {
14           appUtils.init();
15        } catch (Exception e) {
16           e.printStackTrace();
17        }
18        instanceEC2 = appUtils.createInstance(this.getImageId(), this.getName(),
              "t1.micro", "ubuntu", "quicklaunch-1");
19        setHostAddress(instanceEC2.getPublicDnsName());
20        appUtils.waitforInstanceCheck(instanceEC2.getInstanceId(), 200);
21        try {
22           SSHUtils sshHandler = new SSHUtils();
23           Session sshSession = sshHandler.createSSHSession("ubuntu",
                 prop.getProperty("privateKey"),hostAddress);
24           System.out.println("SSH tunnel is openned for "+this.getName()+" at the remote
                 host "+hostAddress);
25           sshHandler.sshRemoteCommand(sshSession, "hostname");
26           sshHandler.sshRemoteCommand(sshSession, "sudo hostname " + hostAddress);
27           ......
28           ......
29           sshSession.disconnect();
30        } catch (IOException e) {
31           e.printStackTrace();
32        }
33        EList<SoftwareComponent> listSoftwareComps = this.getRuns();
34        for (int i=0; i< listSoftwareComps.size();i++){
35           listSoftwareComps.get(i).start();
36        }
37        updatePorts();
38     }
39     @Override
40     public void stopNode(){
41        appUtils.terminateEc2InstanceByID(instanceEC2.getInstanceId());
42     }
43  } //EC2NodeImpl
```

The *stop* operation of a VMI node is represented by the *stopNode* procedure in Listing 4.4. In the *stopNode* procedure, we just call another procedure (named *terminateEc2InstanceByID*) which is defined by the specific APIs in the additional tool - *AmazonEC2Utils* to terminate the running virtual machine.

Listing 4.4: The *stopNode* implementation of the VMINode in case of Amazon EC2

```
1     @Override
2     public void stopNode(){
3        appUtils.terminateEc2InstanceByID(instanceEC2.getInstanceId());
4     }
```

4.3.1.5 SoftwareComponent

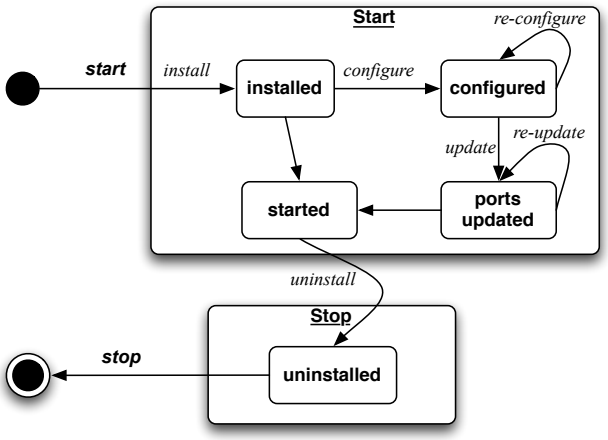


Figure 4.9: Life cycle of a SoftwareComponent instance

The *SoftwareComponent* entity is an abstract presentation of the software package. A *SoftwareComponent* entity defines and encapsulates the instructions of how a software package will be installed or uninstalled. Like the *VMIDeployModel* and *VMINode* entities, it also has two actions: *Start* and *Stop* equivalent to the installing and uninstalling a software package in a virtual machine. The life-cycle of a *SoftwareComponent* instance is represented in the Figure 4.9. It shows the behaviour of a software component. From the figure, we see that a software component can be one of the states: *"installed"*, *"configured"*, *"ports updated"*, *"started"* and *"uninstalled"*. It can respond to the events: *"install"*, *"configure"*, *"re-configure"*, *"update"* and *"uninstall"*. These events are executed sequentially in the order shown in the figure, for example, the events *"configure"*, *"update"* or *"uninstall"* cannot be executed before the event *"install"*.
A software component is started by the *"install"* event. Depending to the specific software, the execution of the software component with the *"install"* event will install the software packages to the running VM. When the installation is finished the state of the software component is transited to *"installed"*. If the software does not to configure the parameters then the state will automatically switch to the *"started"* state. Otherwise, the event *"configure"*, *"update"* will be applied and then the state of the software component will be *"configured"* and *"ports updated"* accordingly. Finally, the state of the software component is transited to *"started"* automatically, and the installed software is ready to use.
The abstract definition of the software component is represented in Listing **??**. It defines the attributes of a software such as *name*, *version*, etc., and two actions (*start* and *stop* procedures): *Start* and *Stop*.

Beacause each software package has specific instructions for the installation, therefore, the *SoftwareComponent* has to have ability to generate different instances that specify different software packages. Figure 4.10 is an example of the *SoftwareComponent* with its children for

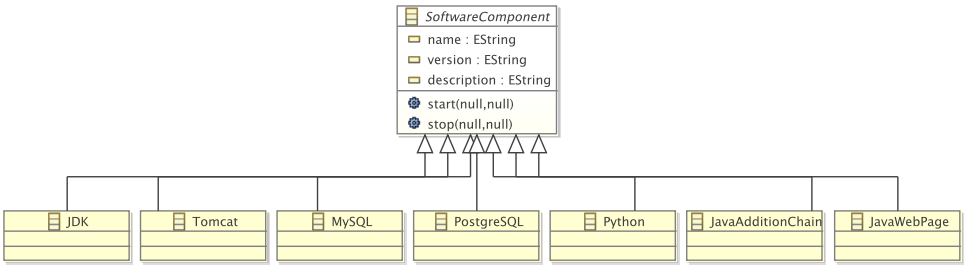


Figure 4.10: An example of the abstract definition of software packages

representing the specific software packages (such as software: *JDK*, *Tomcat*, *MySQL*, etc., or a Java program *JavaAdditionChain*, Java web page *JavaWebPage*). According to the installation and uninstallation instructions of the software, we define the suitable implementation of the corresponding software component to handle its behaviours. Listing 4.5 presents the implementation of a software component for *Java Development Kit (JDK)* installer. In the listing, we define two procedures: *start* and *stop* for handling the installation and uninstallation of *JDK*.

Listing 4.5: The abstract representation of the *SoftwareComponent*

```
1  public class JDKImpl extends SoftwareComponentImpl implements JDK {
2     protected JDKImpl() {
3        super();
4     }
5     @Override
6     public void start(){
7        VMINode hostedNode = this.getRunsOn();
8        String osType = hostedNode.getOsType();
9        SSHUtils sshHandler = new SSHUtils();
10       Properties prop = new
             PropertiesFile().loadProFile(hostedNode.getConfigFilePath());
11       try {
12          Session ssh = sshHandler.createSSHSession("ubuntu",
                prop.getProperty("privateKey"), hostedNode.getHostAddress());
13          if (osType.equals("Linux Ubuntu")){
14             sshHandler.sshRemoteCommand(ssh, "sudo add-apt-repository
                   ppa:webupd8team/java");
15             sshHandler.sshRemoteCommand(ssh, "sudo apt-get -y update");
16             sshHandler.sshRemoteCommand(ssh, "sudo apt-get -y install
                   oracle-jdk7-installer");
17             sshHandler.sshRemoteCommand(ssh, "update-alternatives display java");
18             sshHandler.sshRemoteCommand(ssh, "java -version");
19          }
20          else if (osType.equals("Linux RedHat")){ ... }
21          ...
22       } catch (IOException e) { e.printStackTrace();}
23    }
24    @Override
```

```
25    public void stop(){
26       VMINode hostedNode = this.getRunsOn();
27       SSHUtils sshHandler = new SSHUtils();
28       String osType = hostedNode.getOsType();
29       Properties prop = new
             PropertiesFile().loadProFile(hostedNode.getConfigFilePath());
30       try {
31          Session ssh = sshHandler.createSSHSession("ubuntu",
                prop.getProperty("privateKey"), hostedNode.getHostAddress());
32          if (osType.equals("Linux Ubuntu")){
33             sshHandler.sshRemoteCommand(ssh, "sudo apt-get -y remove
                   oracle-jdk7-installer");
34          }
35          else if (osType.equals("Linux RedHat")){
36             ...
37          }
38          ...
39       } catch (IOException e) { e.printStackTrace();}
40    }
41 } //JDKImpl
```

4.3.2 VMI deployment models

By using the VMIs deployment metamodel, the cloud providers can create the VMIs deployment model on-demand easily. The deployment model of the VMIs in cloud computing can be classified into two types: (i) the deployment model for multiple virtual machines that have the same configuration, and (ii) the deployment model for multiple virtual machines that have the different configurations.

Figure 4.11 is an example of a model for the deployment of three virtual machine with the same configuration. In this example, the model named *Model1* is an instance of the *VMIDeployModel* entity. It has three VMI nodes, that are instances of the *VMINode* entity that represent three machines named: *node 1*, *node 2*, and *node 3*. These machines have the same configuration:

- *imageId* is **ami-6d532204**, this is an ID of a Amazon EC2 image
- *osType* is **LinuxUbuntu**, it specify that the machines run with the Linux Ubung operating system
- *userName* and *password* for all three machine are the same (**admin**/**admin**)
- *configFilePath* is **con.properties**, this file provides additional configuration information for monitoring virtual machines in the specific cloud platform

One thing to note here is that the value of the *hostAddress* attribute is **null**. Because many cloud systems (e.g. Amazon EC2, IBM SmartCloud, OpenNebula, etc.) assign the dynamic IP addresses to the virtual machines. Therefore, in the initial time, the value of *hostAddress* attribute of a VMI model is null, this value will be updated by the IP address or public domain

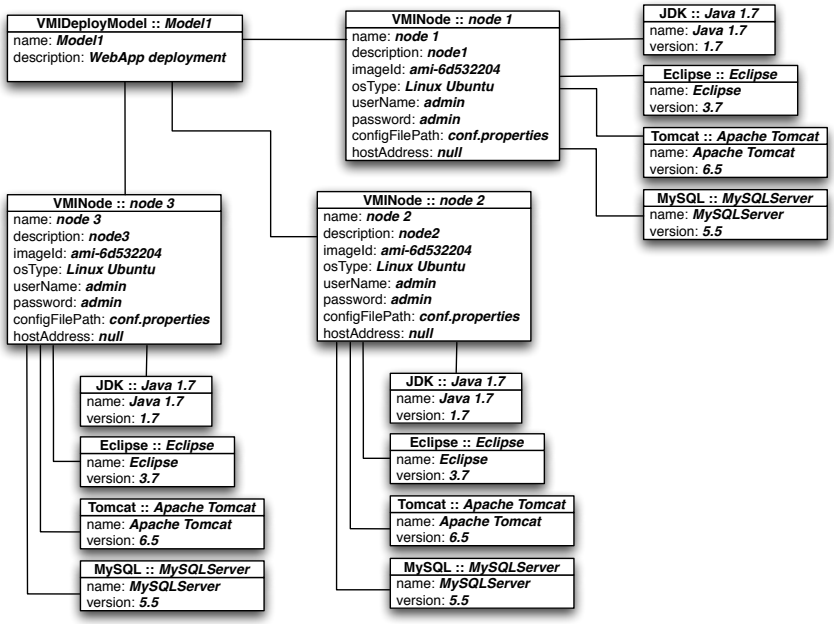


Figure 4.11: An example of a VMI deployment model of multiple VMs with the same configuration

name server (DNS) of the machine when it is deployed successful. For example, the *hostAddress* value can be *ec2-23-23-7-100.compute-1.amazonaws.com*
From the figure, we also see the model specifies that each virtual machine will run the following software packages: *Java 1.7*, *Eclipse*, *Apace Tomcat*, and *MySQL Server*. These software are represented by the elements: *JDK*, *Eclipse*, *Tomcat* and *MySQL* accordingly.

In Figure 4.12, we see an example of the deployment model for multiple machine that have different configurations. The model also defines a complex deployment topology of the VMIs with the connections between the software run on different machines. The model *Model2* shows a model for two virtual machines in a web application deployment scenario. It defines the configuration of two machines named *node 1* and *node 2*, and it specifies how the software in these machines interact together. Like the previous example, the basis configurations of two machines are the same:

- *imageId* is **ami-6d532204**, this is an ID of a Amazon EC2 image
- *osType* is **LinuxUbuntu**, it specifies that the machines run with the Linux Ubuntu operating system
- *userName* and *password* for all three machine are the same (**admin**/**admin**)
- *configFilePath* is **con.properties**, this file provides additional configuration information for monitoring virtual machines in the specific cloud platform

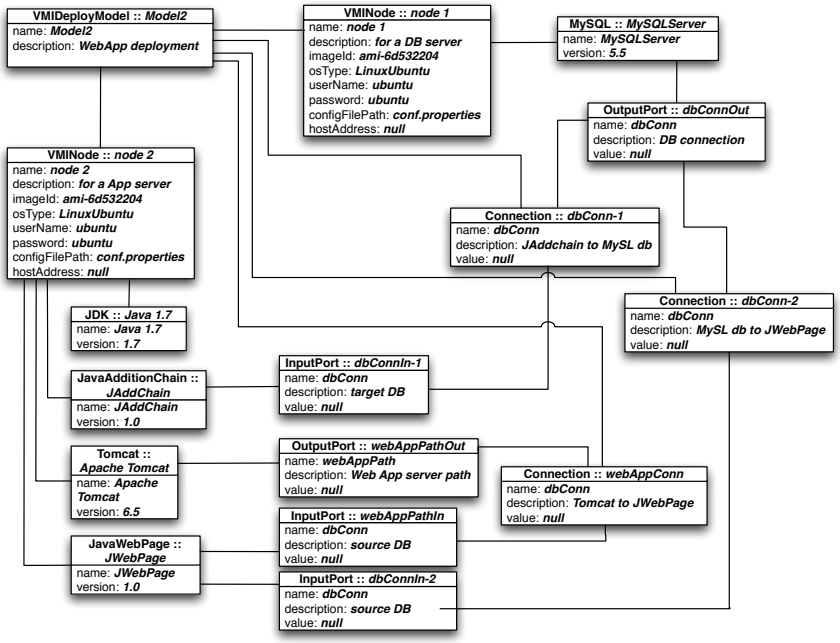


Figure 4.12: An example of a VMI deployment model of multiple VMs with the different configurations

However, the software components running on each machine are different. The virtual machine *node 1* is a database server and contains a software component named *MySQL Server*. This software component is an instance of the *MySQL* entity which represents a *MySQL* database server software, and it defines how this database is installed or uninstalled. This software component has an output port (*OutputPort*) named *dbConnOut*.
The virtual machine *node 2* is an application server. It has four software components that represent the packages that will be installed when the model is deployed:

- *Java 1.7* represents Java Development Kit version 1.7. It provides Java runtime environment for the Tomcat application server and other Java applications
- *JAddChain* is an instance of *JavaAdditionChain* software component model which is a representation of a Java program for calculation the *shortest addition chain numbers*. This programm calculates the chain of numbers and write the results to the database in the MySQL database server. Therefore, it needs information of the target database, such as hostname, port number, name of dabatabase, account to access the database, etc. These information will be received by an input port (*InputPort*) named *dbConnIn-1*
- *Apache Tomcat* represents the Apache Tomcat application server software. It has an output port (*OutputPort*) named *webAppPathOut*. This port used to provide the information of the web application directory path to the other applications
- *JWedPage* is representation of a Java web page what queries the data from a database

in MySQl database server, and displays it on the web page. It needs the information of the source database like the *JAddChain* component, and it also needs the information of the web application directory path to put the web page into the web application server. Therefore, the *JWebPage* software component model has two input ports named *webAppPathIn* and *dbConn-2*

Additionally, we see that the model *Model2* also contains three instances of the *Connection* entity (*dbConn-1*, *dbConn-2*, and *webAppConn*) . These instances represent for the connections between the software components in machines *node 1* and *node 2*.

- *dbConn-1* represents the connection between *JAddChain* program and *MySQLServer* database server by the association of *dbConnOut* port and *dbConnIn-1*
- *dbConn-2* represents the connection between *JWebPage* program and *MySQLServer* database server by the association of *dbConnOut* port and *dbConnIn-2*
- *webAppConn* represents the connection between *JWebPage* program and *Apache Tomcat* application server by the association of *webAppPathIn* port and *webAppPathOut*

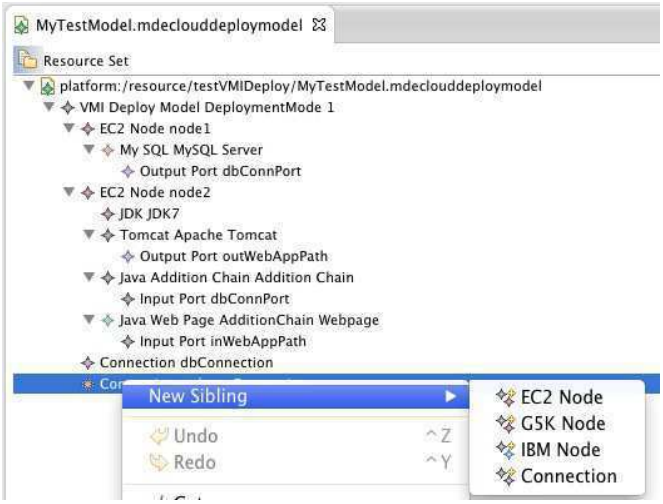


Figure 4.13: An example of a VMI deployment model in EMF editor

The VMIs deployment metamodels are created by the Eclipse Modeling Framework (EMF). It makes easy to generate Java code from the metamodel for the implementing, editing and testing the models. The EMF also provides a graphical user interface environment for creating the VMIs deployment models easily and quickly. Figure 4.13 shows an example of the VMIs deployment model creation in Eclipse treeview editor. The EMF tools also provide the ability to check the validity of the created models based on the constraints defined in the metamodel. Cloud providers can keep the expected VMIs deployment models rather than keeping all virtual machine images that were created for the user 's requests. The Eclipse Modeling Framework allows to store the created VMI deployment model via EMF persistence framework. By defaut, EMF uses XMI (XML Metadata Interchange) format. It is a standard for exchanging metadata

information by XML (Extensible Markup Language). Listing 4.6 is an example of a VMIs deployment model in the XMI format.

Listing 4.6: The abstract representation of the *SoftwareComponent*

```
<?xml version="1.0" encoding="UTF-8"?>
<MDECloudDeployModel:VMIDeployModel xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:MDECloudDeployModel="http://www.kermeta.org/MDECloudDeployModel"
    name="DeployementMode 1" description="The deployment model contain all EC2 nodes">
  <contains xsi:type="MDECloudDeployModel:EC2Node" name="node1" imageId="ami-6d532204"
      userName="admin" password="admin"
      configFilePath="inria.triskell.mdecloud.vmideploymentmodel/ec2ConfigFile.properties">
    <runs xsi:type="MDECloudDeployModel:MySQL" name="MySQL Server" version="5.5"
        description="A database server">
      <outputPort connection="//@connections.0" name="dbConnPort"/>
    </runs>
  </contains>
  <contains xsi:type="MDECloudDeployModel:EC2Node" name="node2" imageId="ami-6d532204"
      userName="admin" password="admin"
      configFilePath="inria.triskell.mdecloud.vmideploymentmodel/ec2ConfigFile.properties">
    <runs xsi:type="MDECloudDeployModel:JDK" name="JDK7" version="1.7"
        description="Java Development Kit"/>
    <runs xsi:type="MDECloudDeployModel:Tomcat" name="Apache Tomcat" version="6"
        description="Apache Tomcat application server">
      <outputPort connection="//@connections.1" name="outWebAppPath" value=""
          description="directory path for web applications"/>
    </runs>
    <runs xsi:type="MDECloudDeployModel:JavaAdditionChain" name="Addition Chain"
        version="1.0" description="A java programm for calculating the addition chain">
      <inputPort name="dbConnPort" connection="//@connections.0"/>
    </runs>
    <runs xsi:type="MDECloudDeployModel:JavaWebPage" name="AdditionChain Webpage">
      <inputPort name="inWebAppPath" value="" connection="//@connections.1"
          description=""/>
    </runs>
  </contains>
  <connections name="dbConnection" inPort="//@contains.1/@runs.2/@inputPort.0"
      outPort="//@contains.0/@runs.0/@outputPort.0" description="The connection
      defines the link between database MySQL and AdditionChain program"/>
  <connections name="webappConnection" inPort="//@contains.1/@runs.3/@inputPort.0"
      outPort="//@contains.1/@runs.1/@outputPort.0" description="the connection
      defines the link between a Java web page and Application Server"/>
</MDECloudDeployModel:VMIDeployModel>
```

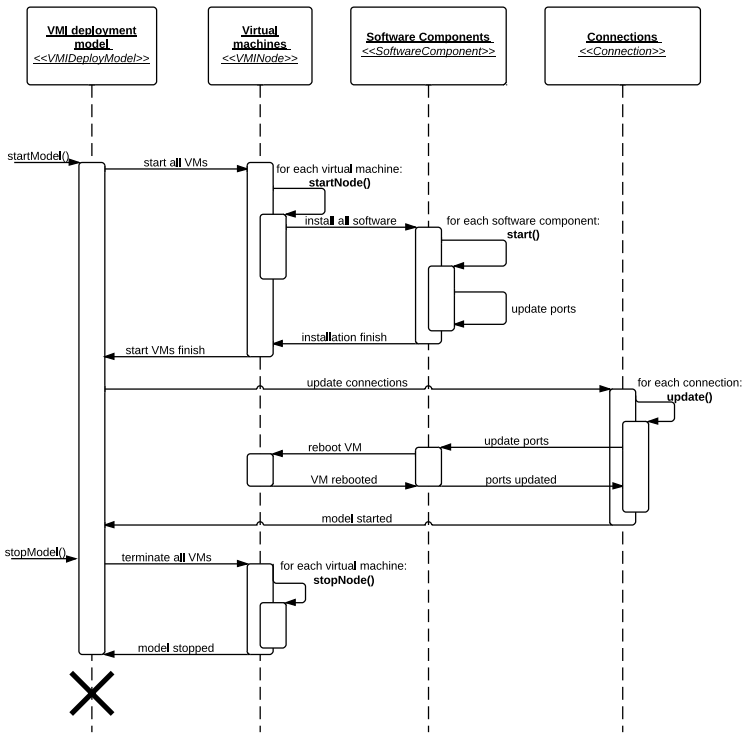


Figure 4.14: Sequence Digram for all steps of the VMIs deployment model execution process

4.3.3 Model execution

The execution of the VMIs deployment model is a sequence of executions of its elements (*VMIDeployModel*, *VMINode*, *SoftwareComponent*, and *Connection*). Figure 4.14 shows a sequence diagram representing how a VMIs deployment model is executed for the VMs provisioning process. The model execution can be considered in two distinct parts: starting the model and stopping the model.

First, the model starts when the user calls the *startModel()* operation, and all the VMI nodes will be considered and executed by the *startNode()* operations, and these processes are executed in parallel. In the execution of *startNode()*, all software components that assigned to the current VMI node will be executed by calling the *start()* operation for the installation and ports configuration. When all the software installations are finished, and VMIs models are started then the execution of the deployment model continues to update the connections. The ports that are linked to the connections will update their values. Sometimes, the updating ports processes also need to reboot their virtual machines. When all connections are updated, the VMIs deployment model is successfully started.

Second, the stopping of the VMIs deployment model happens when the user calls the *stopModel()* operation and then the model will stop by calling the *stopNode()* operation of all virtual machines. The *stopNode()* operation helps to terminate the running machines. When the running machines are successfully terminated, the VMIs deployment model is successfully

stopped.

4.4 The VMIs reconfiguration at runtime process

Another important contribution of using model-driven approach for the VMIs deployment process is providing the ability to reconfigure virtual machine images at runtime. It allows cloud providers to modify the configuration of the running machines to fulfil the change of user 's requirements without creating any new VMIs and re-deploy for replacing the running VMIs. Providers just need to apply the new VMIs deployment model to the current model and the VMIs deployment manager will modify the running system according to the new deployment model. To accomplish these tasks, we have used model@runtime, which is a promising approach for building the adaptive systems. In the next sections, we present more details about how model@runtime is used for the reconfiguration of virtual machine images at runtime.

4.4.1 The model@runtime approach for VMIs reconfiguration at runtime

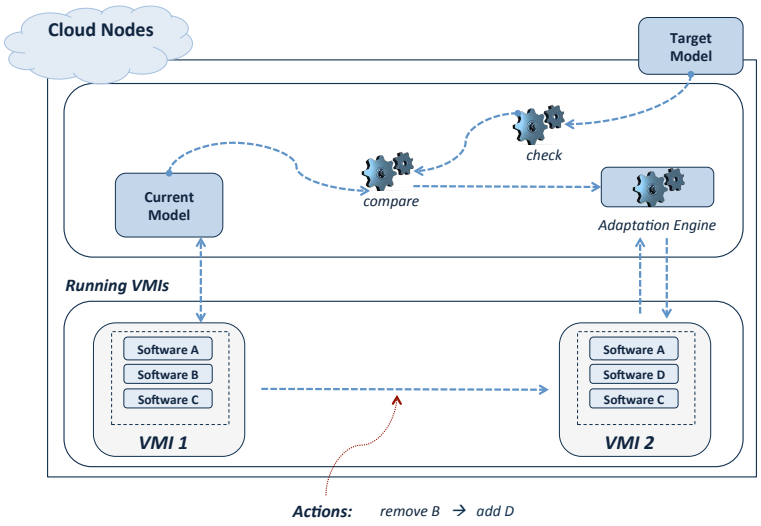


Figure 4.15: Overview of Model@Runtime for managing the changes of the running VMIs

Model@runtime is a promising approach to manage the complexity in the runtime environment by building the adaptation mechanism that leverage the software models [11]. It provides a higher level of abstraction that solves the issue of the system change by reasoning with relevant abstractions of the running system [38]. In the context of VMI provisioning, the Model@runtime technique provides an abstract representation that processes the changes of the running virtual machine images (see Figure 4.15). When the changes appear in the form of a new model (a *target model*) to apply on the system, this new model is checked and validated to malformed configuration for the running VMIs. Then it will be compared with the *current model* which represents the running VMIs. The comparison is processed by a reasoning engine.

This engine determines the changes between models and execute the adaptation steps that bring the current model to the target model.
In Figure 4.15, the *current model* of the running virtual image *VMI 1* contains software packages *A*, *B*, and *C*, the *target model* of the image *VMI 2* contains packages *A*, *D*, and *C*. The running virtual image *VMI 1* is expected to be the image *VMI 2* by some modifications. The reasoning engine compares the *target model* and the *current model* to find the difference of two models and then implement the adaptation steps to change the machine *VM 1* into machine *VM 2*. In the above example, two adaptation steps should be applied to the virtual machine *VM 1*: (i) remove software A, and then (ii) add software D.

4.4.2 The reconfiguration steps

Algorithm 5: An algorithm for updating the configuration of a virtual machine

```
   Input: AdaptationSteps : is a set of the adaptation steps
 1 Procedure UpdateVM(currentVM : VMINode, targetVM : VMINode)
 2     foreach sc_i is a software component of currentVM do
 3         found = false
 4         foreach sn_j is a software component of targetVM do
 5             if sc_i is sn_j then
 6                 found = true
 7                 Break
 8             end
 9         end
10         if found = false then
11             Remove sc_i and its propagated components that found by the
               removePropagated(sc_i) operation from the current virtual machine
12         end
13     end
14     foreach sn_i is a software component of targetVM do
15         found = false
16         foreach sc_j is a software component of currentVM do
17             if sn_i is sc_j then
18                 found = true
19                 Break
20             end
21         end
22         if found = false then
23             Add sn_i and its propagated components that found by the
               addPropagated(sn_i) operation to the current machine
24         end
25     end
26 end
```

The reconfiguration process of the VMIs starts with the comparison between the *current model* and the *target model* and continues with the update of the *current model* according to the

differences between two models. The Algorithms 5 and 6 present the reconfiguration execution within a virtual machine and a VMI deployment model. The details of these algorithms are described later. We assume the existence of the following functions:

- **removePropagated(*S*:*SoftwareComponent*)**: returns a set of software components that depend on software S, and the removal of these components does not conflict with other software component in the *target model*
- **addPropagated(*S*:*SoftwareComponent*)**: returns a set of software components that depend on software S, and the addition of these components does not conflict with other software component in the *current model*

The dependence relationships between software components are represented by using the feature models described in Chapter 3.
The Algorithm 5 presents how to reconfigure the software components of a virutal machine. The reconfiguration process is executed in two steps: (i) removing unneeded software components from the running virtual machine, (ii) adding new software components to the running virtual machine.
Firstly, we examine the software components of the *currentVM* node. If a software appears in both *currentVM* and *targetVM* then we skip it and do nothing (Lines 6-9). The condition in Line 6 check if the software components sc_i of the *currentVM* and sn_j of the *targetVM* are the same (e.g. same name, same version, etc.). Otherwise, if a software component sc_i of the *currentVM* does not appear in the *targetVM* then sc_i and its dependent software which are determined by the function **removePropagated(*sc_i*:*SoftwareComponent*)** will be removed from the running virtual machine. Secondly, we consider the software components of the *targetVM* node. If a software component sn_i of the *targetVM* node does not appear in the *currentVM* node then sn_i and its dependent software which are determined by the function **addPropagated(*sc_i*:*SoftwareComponent*)** will be added to the running virtual machine.
The Algorithm 6 presents the reconfiguration of the virtual machines at runtime by updating the *tartget model* to the running system. The algorithm will update the running system in three steps: (i) removing the virtual machines from *current system*, (ii) adding new virtual machines to the *current system*, and (ii) updating the connections between software components.
Firstly, we compare the *current model* with the *target model* to find the VMI nodes that need to be removed from the *current model*. If a VMI node in the *current model* also appears in the *target model* then it will be compared to the node in the *target model* to recognize any changes of its software components by using the Algorithm 5 (Lines 6-9). If the VMI node in the *current model* does not appear in the *target model* then it needs to be removed, that VMI node will be removed from the *current model*. It means that the corresponding virtual machine which is running in the system will be terminated (Lines 11-14).
Secondly, we compare the *target model* with the *current model* to determine the VMI nodes that need to be added to the *current model*. If a VMI node in the *target model* target model does not appear in the *current model* then it will be added to the *current model*, and then the new virtual machine will be started in the system.
Finally, the reconfiguration process continues with the updating all the connections to establish the new connections or update the value of the ports of connections. When the *target model* is applied successfully to the running system, it is stored as the *current model* of the running system.

Algorithm 6: An algorithm for the reconfiguration of the VMI deployment model

```
1  Procedure UpdateModel(currentModel : VMIDeployModel, targetModel :
   VMIDeployMode)
2      foreach currentVM_i is a VMINode of currentModel do
3          found = false
4          foreach targetVM_j is a VMINode of targetModel do
5              if currentVM_i is targetVM_j then
6                  found = true
7                  UpdateVM(currentVM_i, targetVM_j) Break
8              end
9          end
10         if found = false then
11             Remove the virtual machine sc_i from the current system
12         end
13     end
14     foreach targetVM_i is a VMINode of targetModel do
15         found = false
16         foreach currentVM_j is a VMINode of currentModel do
17             if targetVM_i is currentVM_j then
18                 found = true
19                 Break
20             end
21         end
22         if found = false then
23             Add the virtual machine currentVM_j) to the current system
24         end
25     end
26     UpdateConnection(currentModel, targetModel)
27     currentModel ← targetModel
28 end
```

4.5 Chapter summary

In this chapter, we presented our approach of using model-driven engineering for modeling the deployment and reconfiguration of virtual machine images at runtime. We formalized the software packages, the virtual machine images, the deployment topology of the VMIs as models, and encapsulated the deployment and reconfiguration operations of the virtual machine image into the models. This approach supports the cloud providers to systemize the process of creating and deploying VMIs, and then applying the codified process to manage the deployment and reconfiguration of the images rather than the time-consuming copying of a virtual disk image in the provisioning process. It also provides the ability to design the on-demand deployment scenarios of VMIs on the different cloud platforms.
Additionally, we presented the use of model@runtime technique to support the reconfiguration of the images at runtime. This technique makes more flexible the VMIs provisioning process in

cloud computing.
In summary, this chapter gives solutions for solving the challenges which were addressed in Chapter 1:

- Automating the deployment of VMIs to reduce the deployment time, resources and error-prone
- Supporting the reconfiguration of VMIs and auto-scaling the images at runtime
- Providing a flexible way to hanle the complex tasks of the VMI provisioning process in cloud computing

Part III

Experiment Evaluation & Conclusion

Chapter 5

Experiment Evaluation

Contents

5.1 Chapter Overview

In this chapter we present the results of empirical experiments conducted to evaluate the approach proposed in this thesis. Our experiments are deployed on two virtualization environments: Amazon Elastic Compute Cloud and Grid5000. The goal of the experiments is to evaluate the advantage of using model-driven approach for VMI provisioning in cloud computing, and to show how the approach fulfils the addressed challenges.
In Section 5.2, we present a short introduction of two environments that we used for running the experiments.
Section 5.3 describes the experiment results in the context of: Energy consumption (Section 5.3.1) and VMIs deployment and re-configuration at runtime (Section 5.3.2).
Finally, Section 5.4 discusses about how the experiment evaluation reflexes the challenges and open issues that are addressed in the Chapter 1; and then summarizes the chapter.

5.2 Experiment Environments

For evaluating the proposed approach, we run the evaluation scenarios on two virtualization platforms: Amazon Elastic Compute Cloud (EC2) and Grid5000.

5.2.1 Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (EC2) is the core of Amazon 's cloud computing platform. It allows users to rent virtual computers to run their own applications. It provides the scalable deployment of applications by using a Web service which users can use to access and boot an

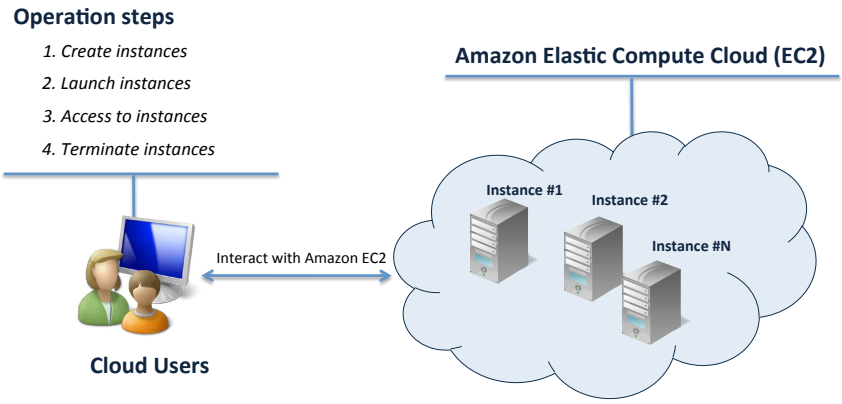


Figure 5.1: Cloud users use Amazon EC2 's services

Amazon Machine Image (AMI), and then install their desired software. Users can create, launch, and terminate the instance on demand, and pay for the usage of running instances (shown in Figure 5.1). Amazon EC2 also allows users to control instances on different geographical locations.
The key features of Amazon EC2 are:

- Providing on-demand computing power. It allows to create and boot new server instances in minutes, and quickly scale capacity up or down;
- Providing different kinds of operating systems (Windows, Linux, FreeBSD, and OpenSolaris);
- Supporting deployment across available zones for reliability;
- Providing the monitoring on status of instances and usage.

Amazon EC2 uses Xen virtualization. Each Amazon EC2 instance is considered as a virtual private server. The size of an instance is based on "Elastic Compute Unit". One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Xeon processor. The cost of renting Amazon instances depends on the instance types which provides the different computational capacity. Figure 5.2 shows the comparison of some Amazon EC2 instances with the computational capacity and cost per hour. In our experiments, we use a basic instance type named as *m1.small*, shaped by the red rectangle.

5.2.2 Grid5000 Virtualization Platform

Grid5000[1] is a virtualization infrastructure for research in France. It includes 9 sites with 19 laboratories involved in France. Grid5000 is designed to support experiment-driven research in all areas of computer science related to parallel, large-scale or distributed computing and networking. It aims at providing a highly reconfigurable, controlable and monitorable experimental platform to its users, and providing the community a testbed allowing experiments in

[1]https://www.grid5000.fr/mediawiki/index.php

API Name	Memory (GiB)	Cores/Compute Units	Instance Storage (GB)	32bit/64bit	I/O Performance	EBS-Optimizable	On-Demand Cost (Linux - per hour on US East 1)
m1.small	1.7	1/1	160	32/64	Moderate	No	$0.060
m1.medium	3.75	1/2	410	32/64	Moderate	No	$0.120
m1.large	7.5	2/4	850	64	Moderate	500 Mbit/s	$0.240
m1.xlarge	15	4/8	1600	64	High	1000 Mbit/s	$0.480
m3.xlarge	15	4/13	0 (EBS only)	64	Moderate	500 Mbit/s	$0.500
m3.2xlarge	30	8/26	0 (EBS Only)	64	High	1000 Mbit/s	$0.480
t1.micro	0.6	1/(up to 2)	0 (EBS Only)	32/64	Low	No	$0.020
m2.xlarge	17.1	2/6.5	420	64	Moderate	No	$0.410
m2.2xlarge	34.2	4/13	850	64	High	500 Mbit/s	$0.820
m2.4xlarge	68.4	8/26	1690	64	High	1000 Mbit/s	$1.640
c1.medium	1.7	2/5	350	32/64	Moderate	No	$0.145
c1.xlarge	7	8/20	1690	64	High	1000 Mbit/s	$0.580

Figure 5.2: Example of the Amazon EC2 image configurations

all the software layers between the network protocols up to the applications.

The Grid5000 sites communicate through a private network. Except for some limited proto-

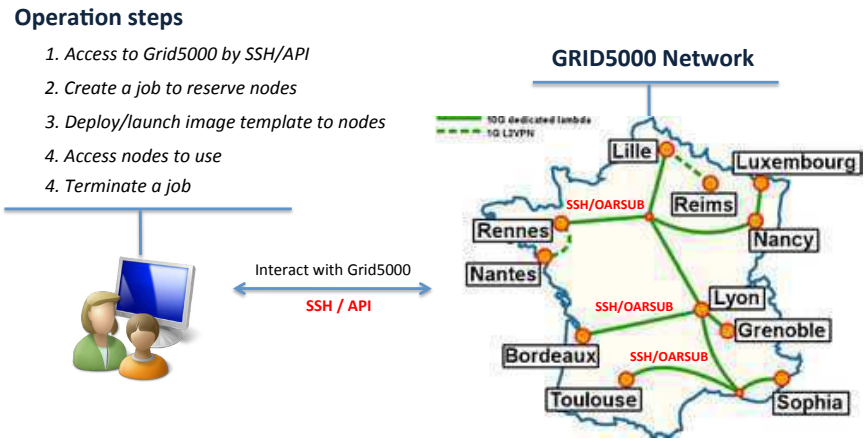


Figure 5.3: Cloud users interact with Grid5000 platform

cols, for instance Ethernet broadcast, there is no limitation in inter site communications. Users can interact with Grid5000 by using SSH protocol or APIs (see Figure 5.3). The users follow some steps to work on Grid5000:

1. Access Grid5000 with the authentication and create a job to reserve the resources (nodes) in a limited time;

2. Launch or deploy template images (provided by Grid5000 or customized by users) into the reserved nodes;

3. Access the nodes and use them (e.g. install software on demand);

4. Terminate the job and release resources

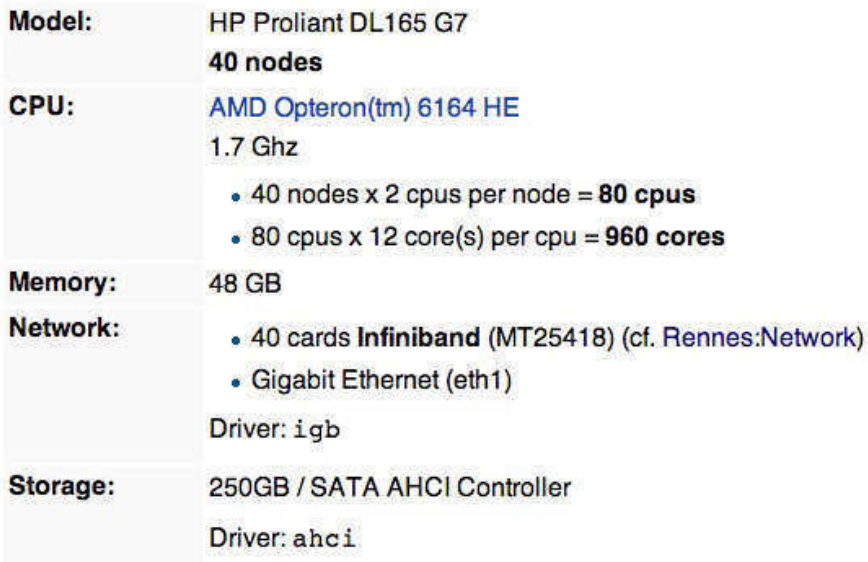


Figure 5.4: The configuration of the *parapluie* cluster at the Rennes site of Grid5000

All virtual machine images supported in Grid5000 run Linux operating system. The hardware configuration depends on the site of Grid5000. For example, at Rennes, there are 3 clusters with different configurations: *parapluie*, *parapide*, and *paradent*. Figure 5.4 is an example of the hardware configuration of the *parapluie* cluster. It has 40 nodes with 2 CPUs per node and 12 cores per CPU, and 48GB memory. Grid5000 also provides tools for users to reserve resources, deploy images, monitor resources, etc.

5.3 Experiment Results

5.3.1 Power consumption comparison

In this section, we present some comparisons between the traditional approach and the model-driven approach in the context of power consumption of VMIs. We consider some key factors that affect the power consumption of VMIs (see Sections 5.3.1.1), and then we show the comparisons of the power consumption of the VMIs by using power consumption meter tools (see Section 5.3.1.2).

5.3.1.1 Data transfer through the network

In terms of power consumption, the amount of data transfer through the network is not as important as I/O or CPU usage. However, it is a factor that needs to be considered for reducing power consumption, since transferring more data implies consuming more network bandwidth [47]. This means that the network equipments (e.g. switches, routers.) consume

more power. Reducing the VMI size helps to reduce the amount of data transfer through the network, so that it reduces the energy consumption of virtual machine transfers. The scenario describes the generation of a VMI that includes selected software stacks in the preceding example (Java, Tomcat, MySQL), and the deployment of this VMI on the Grid5000 reserved nodes. We compare our approach to the traditional approach in terms of amount of data transfer through the network, and power consumption of virtual machine images. We evaluate the traditional approach in two cases:

- Case 1: There is no existing VMI that fits the requirements. The cloud provider needs to create a new VMI containing Java, Tomcat and MySQL.
- Case 2: There is an existing VMI that fits the requirements. It is used as a standard VMI for deploying on the cloud nodes. However, for meeting different user requirements, it also contains software that may not be used: Java, Tomcat, MySQL, Apache2, Jetty, PHP5, Emacs, PostgreSQL, DB2-Express C, Jetty, LibreOffice, etc.

For estimating the amount of data transferred through the network in deployment process, we assume that:
VMI_{T1} : is the size of the clean VMI of case 1 in the traditional approach.
$VMI_{T1'}$: is the size of the clean VMI of case 1 after the installation of the needed software (i.e., Java, Tomcat, MySQL).
VMI_{T2} : is the size of the existing VMI found of case 2 in the traditional approach.
VMI_M : is the size of the standard VMI of model-driven approach.
S : is the size of the needed software installed a VMI in archive format
N : is the number of cloud nodes to deploy

Therefore, the amount of data transfer through the network of both approaches is calculated with the following formulas:

- **Model-driven approach:**
  $DataTransfer = (VMI_M + S) * N$
- **Traditional approach - case 1:**
  $DataTransfer = VMI_{T1} + S + VMI_{T1'} + VMI_{T1'} * N$
- **Traditional approach - case 2:**
  $DataTransfer = VMI_{T2} * N$

⋆Experiment on the Grid5000
In this experiment, we use a clean image **Squeeze-x64-nsf**[2] (333.587 MB), which is available on the Grid5000 's repository. This is also the standard VMI for the case 1 of the traditional approach. In our approach, we use minimal configuration images, only containing an installation software and its dependencies (e.g. Chef, Ganglia, etc.). After the installation of the minimal software, the image size is 339.955 MB. In the case 2, unused software is installed for adapting different requirements from users. This makes the size of a standard VMI is much bigger, 803.60 MB.
Figure 5.5 shows that in both cases, the model-driven approach transfers less data than the

[2]https://www.grid5000.fr/mediawiki/index.php/Squeeze-x64-nfs-1.1

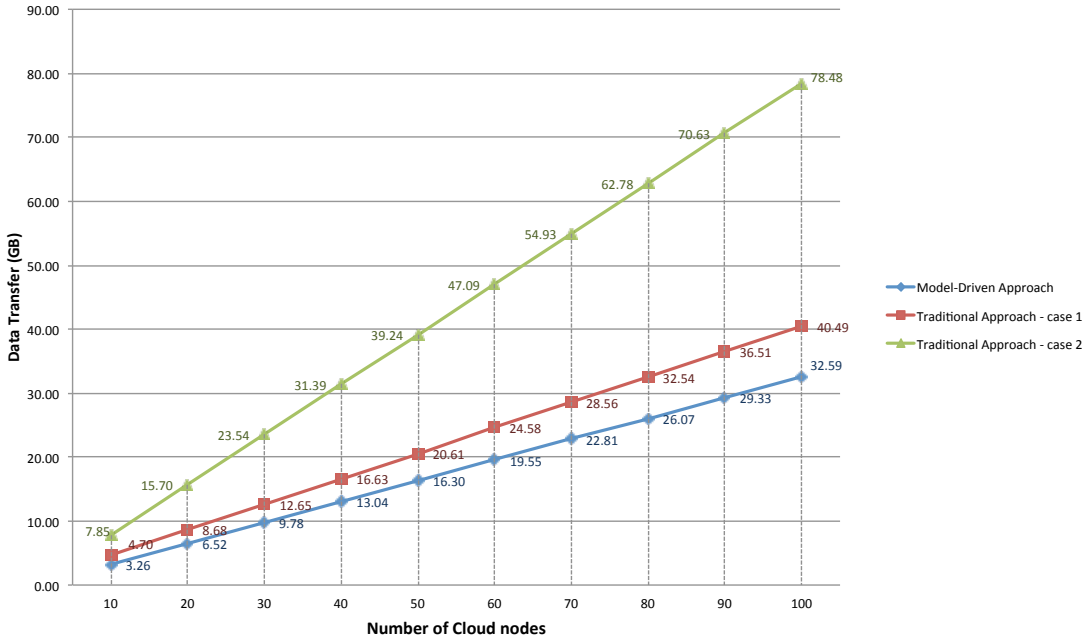


Figure 5.5: Data Transfer Through the Network of the VMI Deployment on Grid5000

traditional approach. Especially when the pre-packaged VMI contains more software installed, and deploys to a large number of cloud nodes. In this example, when we deploy 100 cloud nodes, the traditional approach transfers 40.49GB of data for case 1, and 78.48GB of data for case 2, while the model-driven approach only transfers 32.59GB of data. This result shows that the model-driven approach transfers less data than the traditional approach. It means that the power consumption of network equipment (e.g. switches, routers, etc.) can be reduced.

5.3.1.2 Power consumption

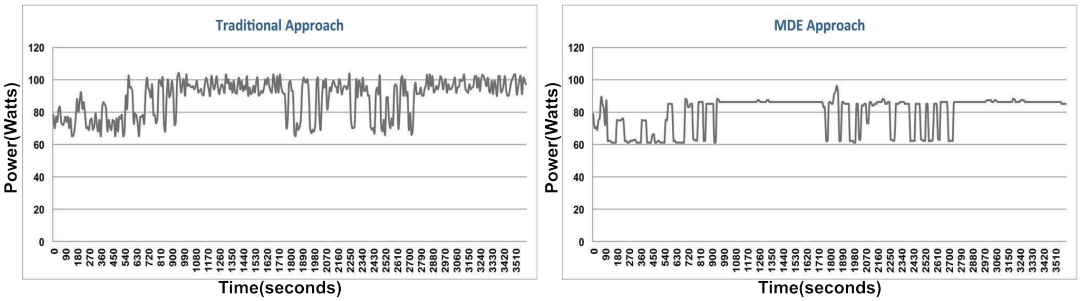


Figure 5.6: Power Measurement from inside the VMIs

We consider the power consumption of the model-driven approach and the traditional approach in two ways: the power measurement from inside the running VMs and the power

consumption of the cloud nodes which host the running VMIs. In our approach, we create and deploy specific VMIs, as in the above example, while the images used for the traditional approach contain the unneeded software. This software is also booted and executed when the VMI is running, meaning that the unneeded software use computing resources (e.g. CPU, RAM.). We simulate programmers doing Java web application programming, showing how the computing resources are used at runtime. In this simulation, we use a script to auto re-compile a Java program and update it to the Tomcat web server. The small Java program emulates some complex deterministic computation by generating the *shortest addition chains*[3] of a number N by a recursive method and writes the results to a MySQL database; the results can be displayed on a JSP web page which is executed by Apache Tomcat web server. We run this program several times with different values for N.

⋆ **Power measurement from inside the running VMIs**
To estimate the power consumption from inside the running VMIs, we deploy these virtual images on the same node of *Graphene* cluster on the Nancy site of Grid5000. In this experiment, we consider two aspects: First, we access the power monitoring hardware installed on this cluster to get information on PDUs (Power Distribution Unit). According to the SNMP MIB[4], description the *outletWatts* give a unique value (in Watts) for the active power sensor attached to the outlet [5]. We consider the power consumption of the running VMIs in 1h with the interval request of 10 seconds. We run this scenario five times and get the mean values of power consumption. After that, we measure the average power consumption of the virtual image in the traditional approach is of **87.43** Watts and this value in the MDE approach is **78.25** Watts. It shows that the image which is created by model-driven approach used less power than the image created by traditional approach.

⋆ **Power measurement of cloud nodes**
We deploy virtual images on the same node *sagittaire-53.lyon.grid5000.fr* of the cluster *sagittaire* on the Lyon site of the Grid5000. To estimate the power consumption of the cloud nodes, we use a live monitoring tool developed by the Green-Net[6] research team. This tool uses wattmeters provided by the SM Omegawatt[7] and it helps to measure the electrical consumption of

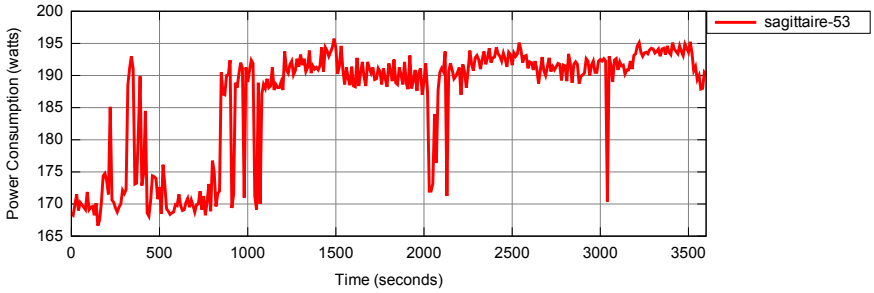


Figure 5.7: Power Measurement of a VMI by the Traditional Approach

[3] http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html
[4] http://www.net-snmp.org/
[5] http://www.grid5000.fr/mediawiki/index.php/Power$_M easurements_i n_N ancy$
[6] http://www.ens-lyon.fr/LIP/RESO/Projects/green-net/
[7] http://omegawatt.fr/gb/index.php

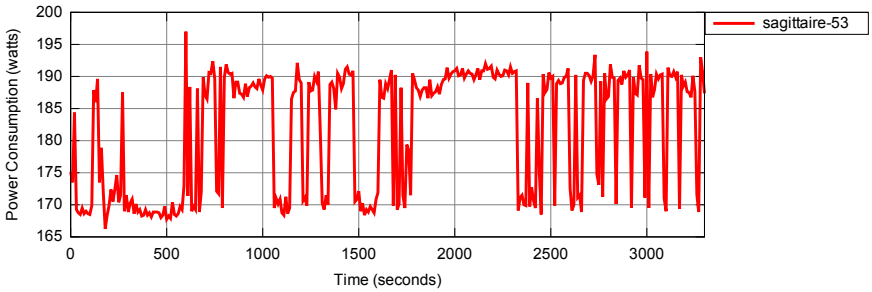


Figure 5.8: Power Measurement of a VMI by the Model-Driven Approach

all nodes on Grid5000 's Lyon site in a real-time manner [31]. Figure 5.8 and Figure 5.7 show the visualization of power consumption of both approaches model-driven and traditional. The experiment results show that the virtual image created by the traditional approach consumes **187.3** Watt-hours, while the image deployed by the model-driven approach consumes **179.6** Watt-hours. We can see that the difference of power consumption of two approaches is not much when we consider only one cloud node within 1h. However, it is much different when we consider on the private cloud system with a large number of nodes running during a long time. Let us think about an example of software companies, they have their own private clouds and provide a platform as a service for hundreds developers working for several months according to their projects. If they have 50 developers working during six months in the above scenario, the model-driven approach helps to reduce from **1.66** to **1.98** Megawatts comparing to the traditional approach. Therefore, the companies can save a considerable amount of electrical expense.
The above two comparisons give different results. This, because the first one is a measurement the power consumption of software running from inside the VMI, while the second one is a measurement of a cluster node that hosts the running VMI. Therefore, it is also influenced by other software used to monitor the running VMI.

5.3.2 VMI re-configuration at runtime

In this section, we present the experiments to show the advantage of using the model-driven approach in the reconfiguration of VMIs at runtime. We consider some typical scenarios of re-configuring the running VMIs. Details of the scenarios are described in Sections 5.3.2.1 and 5.3.2.2.

5.3.2.1 Scenario 1: *Changing the software components of the VMIs at runtime*

In this scenario, we assume that a cloud provider needs to set up a cloud environment with N nodes for a software-development team. All images run on these nodes have the same software configuration. Software developers use these images for testing how a Python program works with different databases MySQL, PostgreSQL, etc. At the runtime, users want to change the MySQL database in the running images by another database (PostgreSQL) for testing.The experiment will show how our approach supports the change of the running VMI configurations according to the user 's demands and the synchronization of the changing to all VMIs easily. The implementation of the scenario is described by the following steps.

⋆ **Step 1:** *Creating VMs with the specific configurations*
In this step, we define a VMI deployment model (named *Model1a*) for two virtual machine

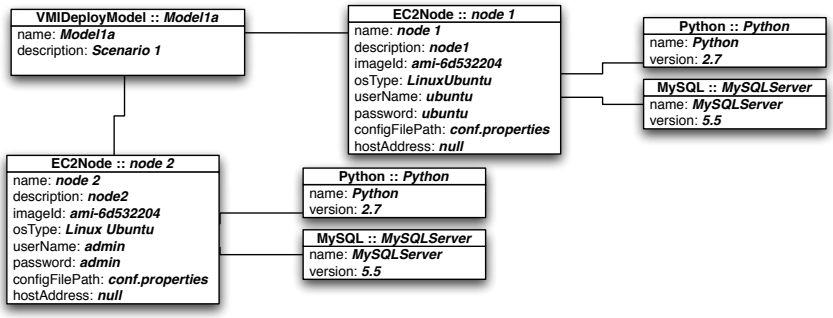


Figure 5.9: VMI deployment model for two EC2 instances that contain *Python* and *PostgreSQL*

images running on the Amazon EC2 cloud platform (shown in Figure 5.9). It contains two VMI models (*EC2Node*) that represents for two nodes (*node 1*, and *node 2*). These VMI models represent for Amazon EC2 instances. We assume that at the beginning, users run machines that contain a Python programming language compiler and a MySQL database server. We use a template image with the Ubuntu 12.04 LTS 64 bits operating system.
After executing the VMI deployment model, we have 2 machines running on Amazon EC2 which are corresponding to 2 EC2 instances with instance IDs: *i-53695b3b* and *i-51695b39*. For validating the installations of software packages, we access manually to these machines and export the list of installed software packages into a file named *step1.txt* by using the following command:

$ dpkg −−get−selections » step1.txt

We will use *step1.txt* file for the comparison between the current configuration and the new configuration of images when we change the software inside the images.

⋆ **Step 2:** *Removing and adding software packages of the running VMs*
In this step, we change the configuration of running images by removing MySQL database and replace it by another database server - PostgreSQL. From the current VMI deployment model *Model1a*(shown in Figure 5.9), we delete *MySQLServer* components and add *PostgresDB* components to the nodes. The new VMI deployment model (named *Model1b*) of running images is described in Figure 5.10. We update this new VMI deployment model to the running system, the VMI Deployment Manager engine recognizes the change between the current model (*Model1a*) and the target model (*Model1b*) and calls the corresponding methods which are defined in the components (*start()* operation for *MySQLServer* and *stop()* operation for *PostgresDB*). After the submission of the target model, the components *MySQLServer* are removed from *node 1* and *node2P*; and then the new components *PostgresDB* are added to the nodes. Therefore, the *uninstallation* operation of *MySQLServer* and the *installation* operation of *PostgresDB* are executed from the nodes. When the execution of the new VMI deployment model is complete, like the Step 1, we access manually to the running machine and export the list of installed software of images into a file name *step2.txt* by using the command:

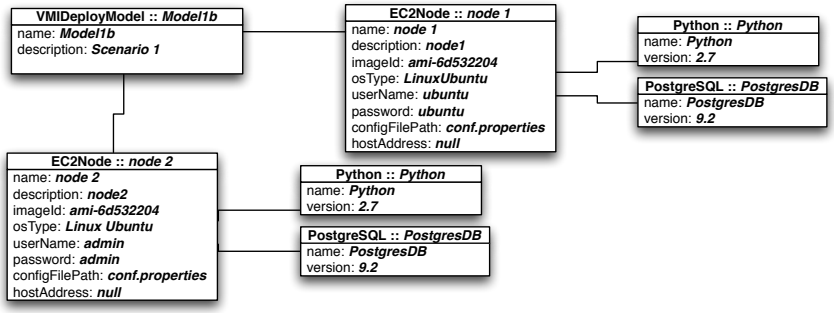
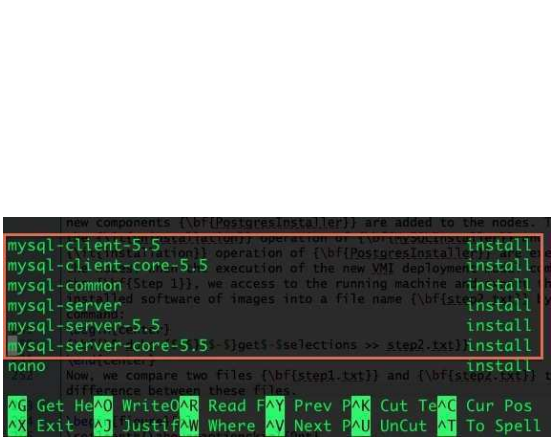
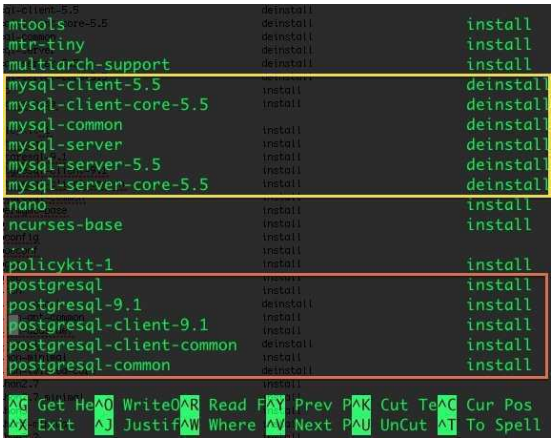


Figure 5.10: The new VMI deployment model for replacing the *PostgreSQL* database by *MySQL* from *Model1a*

$ dpkg −−get−selections » step2.txt



(a) A partial list of the installed MySQL packages in step 1



(b) A partial list of the removed MySQL packages and installed Postgres packages in step 2

Figure 5.11: Example of the installed software package list in two steps

Now, we compare two files *step1.txt* and *step2.txt* from both machines to see the different results between the two steps. From the Figure 5.11, we can see the difference of two files. Figure 5.11a shows the list of MySQL packages, which are installed after the execution in step 1 (in the red rectangle). However, in Figure 5.11b, we can see that the MySQL packages are changed to *deinstall* (in the yellow rectangle), and the Postgres packages are installed (in the red rectangle). It means that the MySQL database is removed from the images, and the PostgreSQL database is installed to the images. By comparing *step1.txt* and *step2.txt* files from both two machines, we also see that the configurations of these machines are the same. It says that the change at runtime occurs with both two VMs, and it guarantees the consistency of VMIs at runtime.

Figure 5.12 is a visual graph of the monitoring on the CPU Utilization of two machines at runtime. The graph is exported by CloudWatch Monitoring tool which is provided by Amazon

Figure 5.12: CPU utilization of two Amazon EC2 nodes at runtime

EC2. The EC2 instances *i-53695b3b*, *i-51695b39* are equivalent to *node 1* and *node 2* in the VMI deployment model, respectively. From the graph, we can see three working stages of VMs. The first stage is booting the machines (creating and launching EC2 instaces). The second stage is the installation of Python and MySQL to the machines, and the changing the running VMIs occurs at the third stage. We also see that two lines of the graph according to the execution of EC2 instances: *i-53695b3b* and *i-51695b39* are similar, it shown that the executions of these instances are the same.

5.3.2.2 Scenario 2: *Changing the deployment topology of the VMIs at runtime*

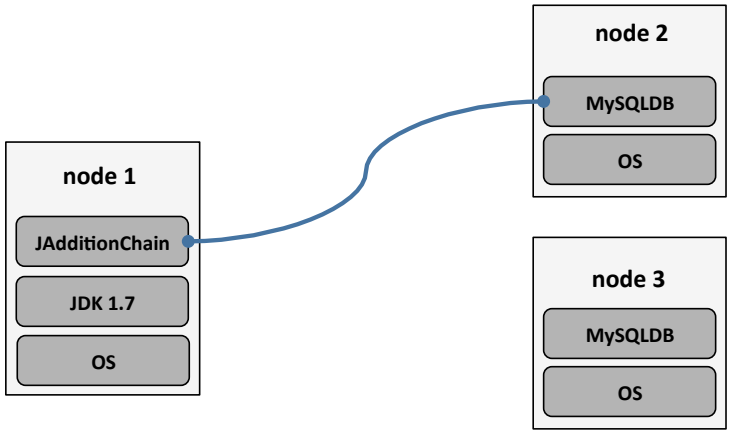


Figure 5.13: Database connection from **node1** points to **node2**

This scenario shows that model-based approach allows the deployment of a complex topology of virtual machine with different configurations. We examine an example of three images to

demonstrate the *shortest addition chain* which is used in Section 5.3.1.2. In this example, we use a machine running the Java program for calculating the *shortest addition chain* of a number N by using a recursive method and writes the result to a MySQL database. Two other machines run the same MySQL database servers. At the runtime, users want to change the database server machine. Then, the experiment shows how model-based approach supports to change the deployment of these VMI easily.
We create a VMI deployment model equivalent to three machines as in Figure 5.13.

Figure 5.13 shows that the database connection from the *JAdditionChain* component on the machine *node 1* is connect to the *MySQLDB* component of the machine *node 2*. When observing the *node 1* component, we notice that there is a component named *JDK 1.7*, which is also added to *node 1* because the VMI Configuration Manager recognizes that it is a dependent software of *JAdditionChain*. A Java program needs Java runtime environment for its execution and *JDK 1.7* is a representative component for the Java runtime environment.
When we execute the VMI deployment model in Figure 5.13, there are three machines (EC2

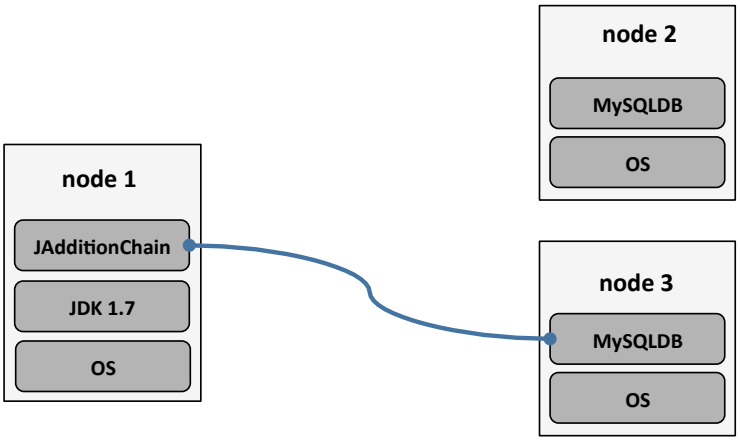


Figure 5.14: Database connection from **node1** points to **node3**

instances) are created and booted on the Amazon EC2 cloud platform. These EC2 instances with the instance IDs: *i-f25be691*, *i-f05be693*, and *i-ae6f66c4* correspond to the three machines: *node 1*, *node 2*, *node 3* respectively. The MySQL database servers are installed in two machines: *node 2* and *node 3*. The *JAdditionChain* component from the machine *node 1* creates the shortest addition chain of the number N, with a number N is automatic created by using randomize function in a range from 200 to 500. For each number N, the result is inserted to the MySQL database that is running on the machine *node 2*.

After the running of the current VMI deployment model in several minutes, we change the database connection of *JAdditionChain* to the MySQL database that is running on the machine *node 3* as in Figure 5.14 and execute the new VMI deployment model. The output results of *JAdditionChain* are inserted into the MySQL database that is running on the machine *node 3* instead of the machine *node 2*. Figure 5.15 shows a visual graph of the monitoring on the CPU utilization of three machines *node 1*(instance id: *i-f25be691*), *node 2*(instance id:*i-f05be693*), and *node 3*(instance id: *i-ae6f66c4*).

From Figure 5.15, we consider the graph as two stages. In the first stage, node 1 and 2 are

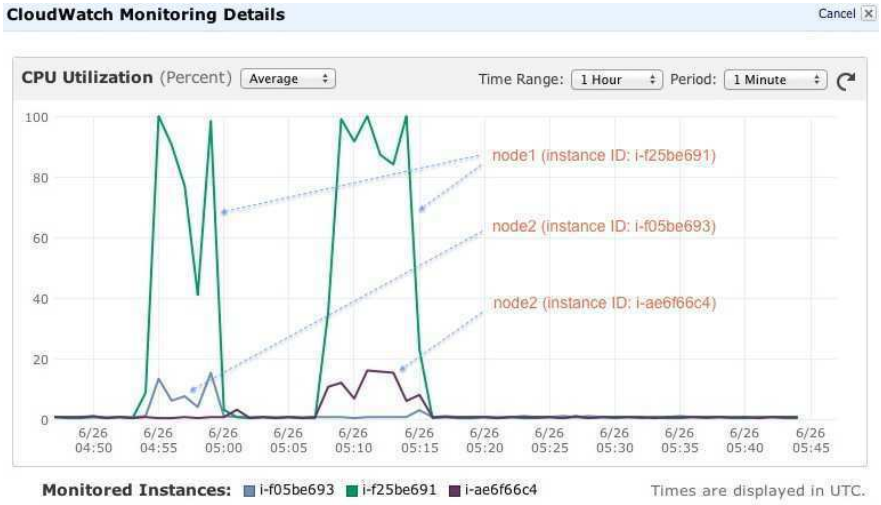


Figure 5.15: CPU utilization of three nodes at runtime

working because the database connection of *JAdditionChain* links to the database on *node 2*. They use a lot of CPU resources while the machine *node 3* is just running with no workload so it uses very few CPU resources.
However, in the second stage, when we change the database connection of *AdditionChain* points to the database on *node 3*; we see that two machines *node 1*, and *node 3* are working and use a lot of CPU resources while the machine *node 2* continues the running with no workload so it uses very few CPU resources.

5.3.2.3 Discussion

From the experiments in the Scenarios 1 and 2, we see that users can benefit from the use of model-driven approach for saving time, reducing the complexity of the VMI provisioning process in cloud computing. By using a graphical user interface, cloud users can create virtual images with on demand configurations and reconfigure them at runtime easily. While in the traditional approach the manipulations of changing the VMI configurations are taken by the experts of the cloud providers, because they require the knowledge of underlying systems and software dependencies.
Table 5.1 is an example of the comparison operations of the VMI reconfiguration at runtime of both approaches: Traditional and Model-driven with using Kevoree framework in the Scenarios 1. In this comparison, the operations of the traditional approach were carried out by IT experts who have the experience and understanding of the virtualization systems and software dependencies.
In the Table 5.1, we see that with each change in the configuration of the running VMIs, the traditional approach takes more time than the model-driven approach. In the above example, it needs 628.423 seconds to change the database from the MySQL to change the Postgres but the MDE approach just needs 197.067 seconds. The main reason leading to this difference stems

Traditional Approach		
Step	Operation	Time (second)
1	Deploy a template VMI to a temporary node	75.082s
2	Install Python	6.854s
3	Install MySQL	48.609s
4	Save running VMI into an image A (AMI)	112.968s
5	Deploy image A to two nodes	81.305s
	Replace MySQL by Postgres	
6	Deploy image A to a temporary node	76.037s
7	Remove MySQL	14.586s
8	Install Postgres	29.663s
9	Save running VMI into an image B (AMI)	105.104s
10	Deploy image B to two nodes	78.215s
Total time		628.423s

MDE Approach		
Step	Operation	Time (second)
1	Deploy a template VMI to two nodes	79.173s
2	Install Python in the running nodes	7.429s
3	Install MySQL in the running nodes	58.038s
	Replace MySQL by Postgres	
4	Remove MySQL in two nodes	17.748s
5	Install Postgres in the running nodes	34.679s
Total time		197.067s

Table 5.1: The comparison of VMI reconfiguration operations of the Traditional approach and Model-driven approach for the Scenario 1

from the traditional approach needs more work to be able to change the configuration of the running VMIs. First of all, it is necessary to build a VMI fits the user's requirements from a certain template VMI (Step 1-3); and then to save the modified VMI into a new VMI (Step 4) for deploying into the nodes (two nodes) as on demand (Step 5). For changing the configuration for the virtual machine is running, the VMI which is modified in the Step 4 will be re-deployed as a temporary virtual machine for changing the configuration accordingly (Step 6-8) and to be saved as a newer VMI for re-deploying to the nodes (Step 9-10).
Additionally, the traditional approach consumes more resources than the model-driven approach, such as: (i) storage resources for keeping as archives the images that are created in the steps 4 and 9; (ii) network resources for transferring the images through the network in the steps 1, 4, 5, 6, 9 and 10 (the estimation of the amount of the data transfer through the network is explained in Section 5.3.1.1).

5.4 Chapter Discussion and Summary

From the experiments in this chapter, we see that the model-driven VMI provisioning process helps to reduce the amount of data transferred through the network, and the power consumption of virtual machines. It provides a mechanism for managing the inter-dependencies of software

Experiment	C1[1]	C2[2]	C3[3]	C4[4]	C5[5]	C6[6]	C7[7]
Data transfer through the network (Session 5.3.1.1)		√					
Power consumption (Session 5.3.1.2)			√				
VMIs reconfiguration in Scenario 1 (Session 5.3.2.1)	√			√	√	√	
VMIs deployment topology reconfiguration in Scenario 2 (Session 5.3.2.2)				√	√	√	√

Table 5.2: How the experiments fulfils the challenges which are addressed in Chapter 1

[1] *C1* - Modeling the variability of VMI 's configuration options to handle the interdependencies of software packages
[2] *C2* - Reducing the amount of data transferred through the networking in the provisioning processes
[3] *C3* - Optimizing the power consumption of VMIs at runtime
[4] *C4* - Providing the graphical interface and easy-to-use tools for user interactions
[5] *C5* - Automating the deployment of VMIs
[6] *C6* - Supporting the reconfiguration of VMIs at runtime
[7] *C7* - Handling the complex and flexible deployment topology of VMIs

packages, and deploying the VMIs with the specific deployment topologies. It also supports for VMIs reconfiguration at runtime to adapt to the user 's requirements. For more details, we examine the comparison on the Table 5.2 to see how the above experiments fulfill the challenges that are addressed in Chapter 1.
In summary, the experiments in this chapter show that the model-driven approach for the VMIs provisioning process in cloud computing provides:

- an abstraction level for managing the configurations of virtual machine images at design time and runtime;
- an abstraction level for handling the deployment process and topology of virtual machine images at design time and runtime.

Chapter 6

Conclusion

Contents

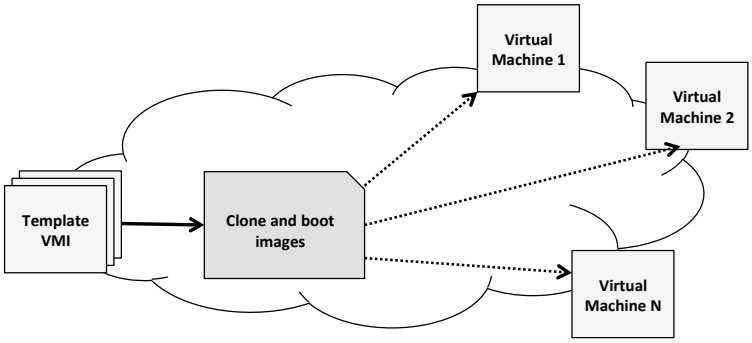
6.1 Conclusion

In this thesis, we proposed a Model-Driven approach for virtual machine images provisioning in cloud computing. We used *feature modeling technique* to manage the configurations of virtual machines, and then we used *model-based methodology* to model and define the processes of virtual machine image deployments and reconfigurations at runtime. We have shown that the model-driven approach improves the performance of the provisioning process and makes the management of virtual machine images to be more flexible and easier than the traditional approach. We also have presented how our approach differs from the traditional approach. A visual comparison of the approaches shown in Figure 6.1. The key difference of our approach compared to the traditional approach can be summarized as the following:

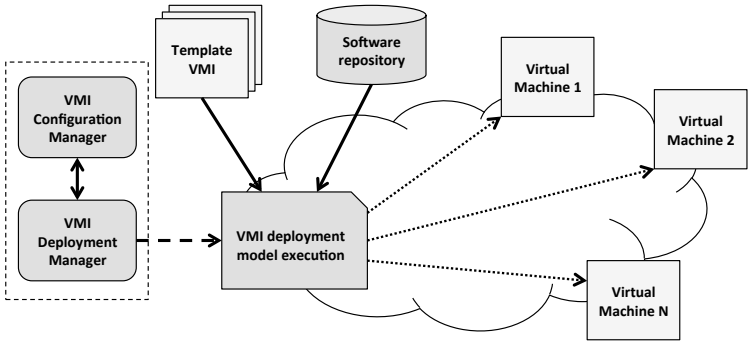
- **Traditional approach:** A standard virtual image, containing all possible software, can be cloned and booted many times
- **Model-driven approach:** Considering the virtual machine images, the software packages and the deployment topology of VMIs as the models; creating the configuration of VMIs and the deployment model at *design time* while creating the concrete VMIs at *runtime*

To conclude the thesis, let us recall our contributions that positioning with respect to the challenges addressed in the introduction chapter (Session 1.3, Chapter 1):

- *C1 - Modeling the variability of the VMI configurations and handling the software package dependencies?*
  We represented a software package as a future element in a feature model. Thanks to the benefit of using feature models for representing the complex dependencies of elements, we can define a software package with its required packages or its mutual exclusive software package. By using the selection algorithms, the feature model helps to select the software packages with their dependencies easily and efficiently. We considered the VMI



(a) Traditional approach



(b) Model-driven approach

Figure 6.1: Traditional and Model-driven approaches for VMI provisioning in cloud computing

configurations as Product Lines, and used feature models to represent the VMI configuration options. The VMI Product Lines provided the ability of analyzing and modeling the commonalities and variabilities of VMIs. The commonality of the VMIs provides the common platforms that contain the base components for determining the characteristics of the VMIs. While the variability represents the flexibility of VMI configurations, it provides the pre-definitions of what possible software shall be installed into a virtual machine, and defines exactly the places where the VMI can differ from the others. We also extended the standard feature model to have ability to represent software and its information (e.g installation time, uninstallation time, size of package, etc.) as the features of the feature model. We used a feature reasoning engine - SPLAR for managing the product derivation process. Like other standard feature reasoning engine, it supports to derive the valid VMI configuration from the user 's selection. We also extended SPLAR by adding the algorithms to adapt to the requirements of the product derivation, for example finding the optimal VMI configuration in the term of minimum installation time.

- *C2 - Reducing the amount of data transferred through the network in the provisioning process*
  In our approach, we create the expected virtual images at runtime, when the template VMI already deployed on the cloud nodes. The template VMIs are the images with

minimum configuration, and their sizes are smaller than the images that contain many kinds of software packages, so the amount of data transferred through the network of these template images is less than the amount of data transferred to the case of images contain unneeded software.

- *C3 - Optimizing the power consumption of VMIs at runtime*
  By using the VMI feature models, our approach creates the expected VMs that contain only the needed software packages and their dependencies. They do not have unnecessary software to run during the operation, so they will save the computing resources (CPU, memory, storage, etc.) and the power consumption of the virtual machines.

- *C4 - Providing the graphical interface and easy-to-use tools for user interactions*
  We developed a prototype framework with graphical user interfaces for the management of VMI configuration and deployment processes.
  We used the feature model reasoning engine and developed a graphical user interface for the interaction between users and the reasoning engine. Therefore, the users can create VMI configurations on demand by selecting the needed software packages while they do not need to care about the required or the mutual exclusive packages because these packages are selected or de-selected automatically by the reasoning engine.
  We built a metamodel to pre-define the constructions and rules for the creation of the VMI deployment model by using Eclipse Modeling Framework, so the users can create the VMI deployment models easily and avoid errors occurred during the operating.

- *C5 - Automating the deployment of VMIs*
  We represented the deployment scenarios, the virtual machines and software packages as the models, and the models encapsulated the pre-definitions and the behaviours of these things work. We define the models with two key operations: *start* and *stop*. The *start* operation defines how a model is executed, for example, how a virtual machine start and install the needed software packages, while the *stop* operation defines the way to terminate the running model, for example, it defines how to remove a specific software from a virtual machine image and how to shut-down the running machine, etc. We develop a component to handle the execution of the models (VMI deployment models, VMI models and software component models). It help to manage the image deployment with the expected configuration automatically.

- *C6 - Supporting the reconfiguration of VMIs at runtime*
  We used a model@runtime approach for implementing the deployment and reconfiguration of the virtual machine images. We manage the deployment and reconfiguration of the image and software packages through the model that encapsulate the pre-definitions and the behaviour of the models. A VMI deployment model represents a topology of a VMI deployment scenario. It also defines the virtual machine images with the corresponding software packages. When the VMI deployment model is executed, the virtual machine images are booted and the corresponding software packages are installed into the virtual machines. For scaling (add new or remove) the images or reconfiguring the running images, the new deployment model that represents for the new deployment scenario will be compared to the current model which is represent the running system to find the difference between two models. After that, the reconfiguration steps are applied to the running

system to change it into the new one that corresponding to the new VMI deployment model.

- *Handling the complex and flexible deployment topology of VMIs*
  We manage the deployment of the virtual machine images through the VMI models. These models provide the abstract representations of the cloud nodes in cloud computing. According to the specific technological of the cloud platforms, the VMI models define the corresponding abstractions and representations for executing the virtual machine images. The deployment of virtual machine images relies on the VMI deployment metamodel which contains the pre-definitions of the constructions and the rules for the model creations. The VMI model is an entity of the VMI deployment metamodel, and when a VMI deployment model is created, it contains the VMI model instances which are represent for cloud nodes in the specific cloud platforms. It ensures that the deployment of virtual machine image can be executed on the federated cloud system.

The partial results of this thesis were presented in international peer-reviewed conferences and published in:

1. Tam Le Nhan, Gerson Sunyé, Jean-Marc Jézéquel. *A Model-Driven Approach for Virtual Machine Image Provisioning in Cloud Computing.* European Conference on Service-Oriented and Cloud Computing, ESOCC 2012. Bertinoro, Italy. Springer ISBN 978-3-642-33426-9

2. Tam Le Nhan, Gerson Sunyé, Jean-Marc Jézéquel. *A Model-Based Approach for Optimizing Power Consumption of IaaS.* IEEE Second Symposium on Network Cloud Computing and Applications, NCCA 2012, London, UK. IEEE Computer Society 2012 ISBN 978-1-4673-5581-0

6.2 Limitations

Although our approach has brought improvements compared to the traditional approach, it still has some limitations. First, it requires to create the VMI feature model for representing the VMI configuration options (software packages). Second, it needs to adjust the algorithms of the feature reasoning engine for finding the optimal configuration with respect to the expected factors. Third, it requires to define the implementations of all software components that represent the software packages and different types of VMI models that represent the virtual machines in the specific cloud platform.

- **Creating the VMI feature model**
  One of the most important steps of representing the variability of the VMI configurations is creating the VMI feature model which is used to represent all the VMI configuration options (w.r.t software packages) and their relationships. The validity of the created VMI configuration relies on the VMI feature model, therefore the creation of VMI feature model must fulfil the constraints for the feature model constructions and software packages. Therefore, the creation of the VMI feature model should be done by experts who have experience and understanding of software and underlying systems.

- **Adjusting the algorithms for the feature reasoning engine**
  The standard feature reasoning engine can support to validate the user 's choices and select the dependent features or deselect the mutual exclusive features. However, the optimal VMI configuration relies on the search factors, such as minimum installation time, operational cost, etc. Therefore, it needs to adjust the search algorithms of the feature model reasoning engine according to the search factors. Additionally, the searching the optimal VMI configuration on multiple search factors is much more complex than the searching on the single search factor. Our approach supports to find the optimal VMI configuration on the single search factor only.

- **Creating VMI models and software component models**
  In our approach, we define the abstract definitions and execution behaviours of virtual machines in different cloud platforms and software packages in the models. Therefore, one needs to create all the VMI models and software component models with respect to the software package that represented in the VMI feature models and the virtual machine in the specific cloud platforms. Because the software packages are diverse and have many kinds, while each software needs corresponding software component model to demonstrate its execution through the two operations: *start* and *stop*. Therefore, it is challenge to develop all software components for the large number of software packages.

6.3 Perspectives

Nowadays, cloud computing is an emerging technology. There are many vendors developing and providing cloud services and infrastructures with different technological platforms. Model-driven engineering have the advantage of providing the high-level abstractions of the systems is a promising approach for cloud computing. Not only solving the problems addressed in this thesis, it also opens up some other research challenges as the following:

- **Managing the variability of cloud services**
  One of the key characteristics of cloud computing is the ability to provide services on demand. The services offered may be the platforms (the virtual machines) as mentioned in this thesis, may also be other computing utilities, such as software, databases, or storages as services. The requirements from users are diverse, but they also have some common features as well as the individual characteristics. Thus recognizing the services offered by cloud computing as a family of products with the analysis of the common platform and the variability of the products will be an effective approach. It will enable the cloud service providers to manage, prepare the services more efficiently, and to save the costs and resources while still fulfill the user's requirements.

- **Managing the cloud service level agreements (SLAs)**
  Cloud users use services and pay for what they used. Therefore, SLA contracts are very important in cloud computing business. A SLA contract defines the rights and responsibilities of both parties: Service providers and service users. The individual customers or groups of customers using the service will have different SLA contracts. However, in the classification of these contracts, we see that the contracts may have similar terms or characteristics if they are used for the customers in the same groups (e.g Silver, Gold, or Titanium groups), or for the groups of similar services (e.g. Network, Storage, CRM,

or ERP services). This suggests that the Product Line engineering is a potential approach for the management of SLA contracts. It allows the cloud providers can create and manage the SLA contracts easily and flexibility.

- **Building adaptive SLA contracts**
  Another promising research direction of the use of model@runtime for the reconfiguration at runtime is building adaptive SLA contracts which are applied to monitor the quality of services (QoS) and then reflect to the SLA contracts. If the actual quality of services conflicts with the terms that mentioned in the contract then the SLA contracts can be adjusted to ensure the rights of the parties. Conversely, if the process is carried and SLA contracts are changed as required, it will reflect the running system to corresponding changes.

- **Transferring, migrating services between different cloud providers**
  Extending out from the model-based applications for deployment and reconfiguration at run-time services that were mentioned in Chapter 4, the model-based approach could be well adapted for the migrating or transferring services between different cloud systems. In this context, the cloud services can be considered as the models which are platform-independent service models. These service models represent the pre-definitions, execution methods of the services at the high-level of abstractions. Thus the migration or transfer of services between different cloud system will be easier and more flexible.

Bibliography

[1] Amazon elastic compute cloud. http://aws.amazon.com/ec2/. (Cited on page 19.)

[2] Ibm smartcloud. http://www-935.ibm.com/services/us/en/cloud-enterprise/. (Cited on page 19.)

[3] Model-driven application deployment for cloud computing environments. White Paper, Sun Microsystem Inc., January 2010. Available online (18 pages). (Cited on pages 4, 20, 33 and 34.)

[4] M. Acher. *Managing Multiple Feature Models: Foundations, Language and Applications*. PhD thesis, Université de Nice–Sophia Antipolis, 2011. (Cited on page 25.)

[5] M. Acher, P. Collet, P. Lahire, and R. B. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 2012. (Cited on page 25.)

[6] N. Antonopoulos and L. Gillam. *Cloud Computing: Principles, Systems and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010. (Cited on pages 15 and 18.)

[7] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009. (Cited on pages 3, 12 and 14.)

[8] W. Arnold, T. Eilam, M.H. Kalantar, A.V. Konstantinou, and A. Totok. Automatic realization of SOA deployment patterns in distributed environments. In *Service-Oriented Computing - ICSOC 2008, 6th International Conference, Sydney, Australia, December 1-5, 2008. Proceedings*, pages 162–179, 2008. (Cited on pages 33 and 34.)

[9] D. Benavides, S. Segura, and R.C Antonio. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6), 2010. (Cited on pages xv and 25.)

[10] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. *Lecture Notes in Computer Science*, 3520:381–390, 2005. (Cited on pages 8, 25 and 42.)

[11] G. Blair, N. Bencomo, and R.B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009. (Cited on pages 22 and 75.)

[12] A. Brown. An introduction to model driven architecture. IBM developerWorks, 2004. http://www.ibm.com/developerworks/rational/library/3100.html. (Cited on page 21.)

[13] R. Buyya, J. Broberg, and A.M. Goscinski. *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011. (Cited on pages 16 and 18.)

[14] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009. (Cited on pages 3 and 12.)

[15] T.C. Chieu, A. Mohindra, A. Karve, and A. Segal. Solution-based deployment of complex application services on a cloud. In *Service Operations and Logistics and Informatics (SOLI), 2010 IEEE International Conference on*, pages 282 –287, july 2010. (Cited on pages 33 and 34.)

[16] T.C. Chieu, A. Mohindra, A.A. Karve, and A. Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on*, pages 281 –286, oct. 2009. (Cited on pages 4, 33 and 34.)

[17] K. Christian, T. Thomas, S. Gunter, F. Janet, L. Thomas, W. Fabian, and A. Sven. Featureide: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 611–614, Washington, DC, USA, 2009. IEEE Computer Society. (Cited on page 25.)

[18] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 3rd edition, 2001. (Cited on page 22.)

[19] K. Czarnecki and U.W Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. (Cited on pages 20, 22 and 23.)

[20] B. Dougherty, J. White, and D.C. Schmidt. Model-driven auto-scaling of green cloud computing infrastructure. *Future Generation Computer Systems*, 28(2):371 – 378, 2012. (Cited on pages 27, 30 and 34.)

[21] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 20.)

[22] R.P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, pages 34–45, June 1974. (Cited on pages 16 and 25.)

[23] R. Han, L. Guo, Y. Guo, and S. He. A deployment platform for dynamically scaling applications in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 506 –510, 29 2011-dec. 1 2011. (Cited on page 33.)

[24] David Hilley and David Hilley. Cloud computing: A taxonomy of platform and infrastructure-level offerings, 2009. (Cited on page 12.)

[25] C.N. Hfer and G. Karagiannis. Cloud computing services: taxonomy and comparison. *Journal of Internet Services and Applications*, 2:81–94, 2011. (Cited on pages xv and 14.)

[26] J.M. Jézéquel. Modeling and aspect weaving. In *MMOSS*, 2006. (Cited on page 20.)

[27] J.M. Jézéquel. Model-Driven Engineering for Software Product Lines. *ISRN Software Engineering*, 2012, 2012. (Cited on pages 22 and 23.)

[28] K. C. Kang, S.G. Cohen, J.A Hess, W.E Novak, and A.S Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990. (Cited on page 24.)

[29] A. Konstantinou, T. Eilam, M. Kalantar, A. Totok, W. Arnold, and E. Snible. An architecture for virtual solution composition and deployment in infrastructure clouds. In *Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing*, VTDC '09, pages 9–18, New York, NY, USA, 2009. ACM. (Cited on pages 33 and 34.)

[30] J. Ludewig. Models in software engineering - an introduction. *Software and Systems Modeling*, 2(1):5–14, March 2003. (Cited on page 20.)

[31] D.A. Marcos, J.P Gelas, L. Laurent, and O. Anne-Cécile. The Green Grid5000: Instrumenting a Grid with Energy Sensors. In *INGRID'2010 : 5th International Workshop on Distributed Cooperative Laboratories: Instrumenting the Grid*, pages 25–42. Springer, 2012. (Cited on page 90.)

[32] P. Mell and T. Grance. The nist definition of cloud computing. Technical report, National Institute of Standard and Technology - NIST, 2011. (Cited on pages 3, 13, 14 and 18.)

[33] M. Mendonça. *Ecient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, 2011. (Cited on pages 42 and 44.)

[34] M. Mendonça, M. Branco, and D.D. Cowan. S.p.l.o.t.: Software product lines online tools. In *OOPSLA Companion*, pages 761–762, 2009. (Cited on pages 25 and 44.)

[35] M. Mendonça and D.D. Cowan. Decision-making coordination and efficient reasoning techniques for feature-based configuration. *Sci. Comput. Program.*, 75(5):311–332, 2010. (Cited on page 44.)

[36] M. Mendonça, A. Wasowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *13th International Conference on Software Product Lines (SPLC 2009), San Francisco, CA, USA*, 2009. (Cited on pages 8, 25 and 42.)

[37] M. Mendonça, A. Wasowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *SPLC*, pages 231–240, 2009. (Cited on pages 25 and 44.)

[38] B. Morin, O. Barais, J.M. Jézéquel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *IEEE Computer*, 42:44–51, 2009. (Cited on pages 22 and 75.)

[39] B. Morin, O. Barais, G. Nain, and J.M. Jézéquel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society. (Cited on page 22.)

[40] K. Pohl, G. Böckle, and F.J. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. (Cited on pages xv, 22, 23 and 28.)

[41] L. Qian, Z. Luo, Y. Du, and L. Guo. Cloud computing: An overview. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 626–631, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on page 12.)

[42] D.C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006. (Cited on pages 21 and 22.)

[43] M. Sethi, K. Kannan, N. Sachindran, and M. Gupta. Rapid deployment of SOA solutions via automated image replication and reconfiguration. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 1, pages 155 –162, july 2008. (Cited on pages 33 and 34.)

[44] J.E. Smith and N. Ravi. An Overview of Virtual Machine Architectures. *Elsevier Science*, November, 2003. (Cited on page 25.)

[45] T. Thüm, D.S. Batory, and C. Kästner. Reasoning about edits to feature models. In *ICSE*, pages 254–264, 2009. (Cited on pages 8, 25 and 42.)

[46] J.P. Tolvanen. *Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence*. PhD thesis, University of Jyväskylä, 1998. (Cited on pages xv and 20.)

[47] A.J. Younge, G. Laszewski, L. Wang, S. Lopez-Alarcon, and W. Carithers. Efficient resource management for cloud computing environments. *International Conference on Green Computing*, 0:357–364, 2010. (Cited on page 86.)

[48] T. Zhang, Z. Du, Y. Chen, X. Ji, and X. Wang. Typical virtual appliances: An optimized mechanism for virtual appliances provisioning and management. *Journal of Systems and Software*, 84(3):377 – 387, 2011. (Cited on pages 20 and 34.)

[49] T. Ziadi, J.M. Jézéquel, and F. Fondement. Product Line Derivation with UML. In *Proceedings of 1st Workshop on Software Variability Management at 25th International Conference on Software Engineering*, 2003. (Cited on pages 22 and 23.)